Chapter 10
Preserve Environment via Stack

Z. Gu

Fall 2025

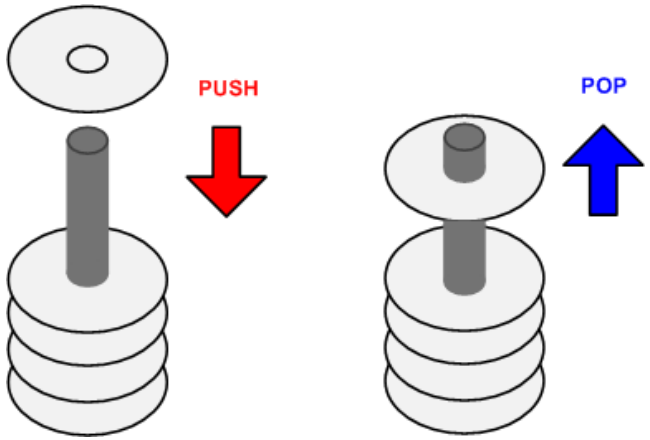
Acknowledgement: Lecture slides based on Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C, University of Maine https://web.eece.maine.edu/~zhu/book/

Overview

- How to call a subroutine?
- How to return the control back to the caller?
- How to pass arguments into a subroutine?
- How to return a value in a subroutine?
- How to preserve the running environment for the caller?

Stack

- A **Last-In-First-Out** memory model
- Only allow to access the most recently added item
  - Also called the top of the stack
- Key operations:
  - **push** (add item to stack)
  - **pop** (remove top item from stack)



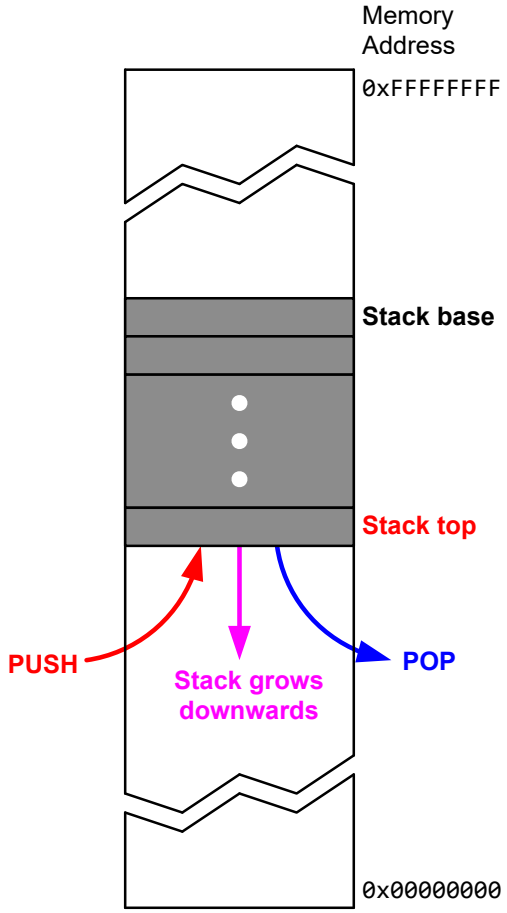
- Example: Tower of Hanoi
  - Only one disk may be moved at a time.
  - Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
  - **No disk may be placed on top of a smaller disk.**

http://en.wikipedia.org/wiki/File:Tower_of_Hanoi_4.gif

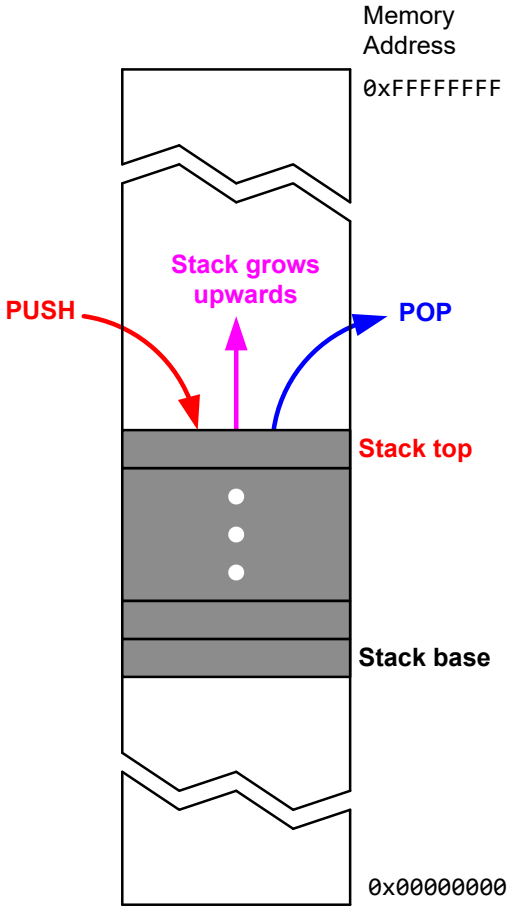
Typical Usage of Stack

- Why need stack?
  - Saving the original contents of processor's registers at the beginning a subroutine (Contents are restored at the end of a subroutine)
  - Storing local variables in a subroutine
  - Passing extra arguments to a subroutine
  - Saving processor's registers upon an interrupt

Stack Growth Convention: Ascending *vs* Descending

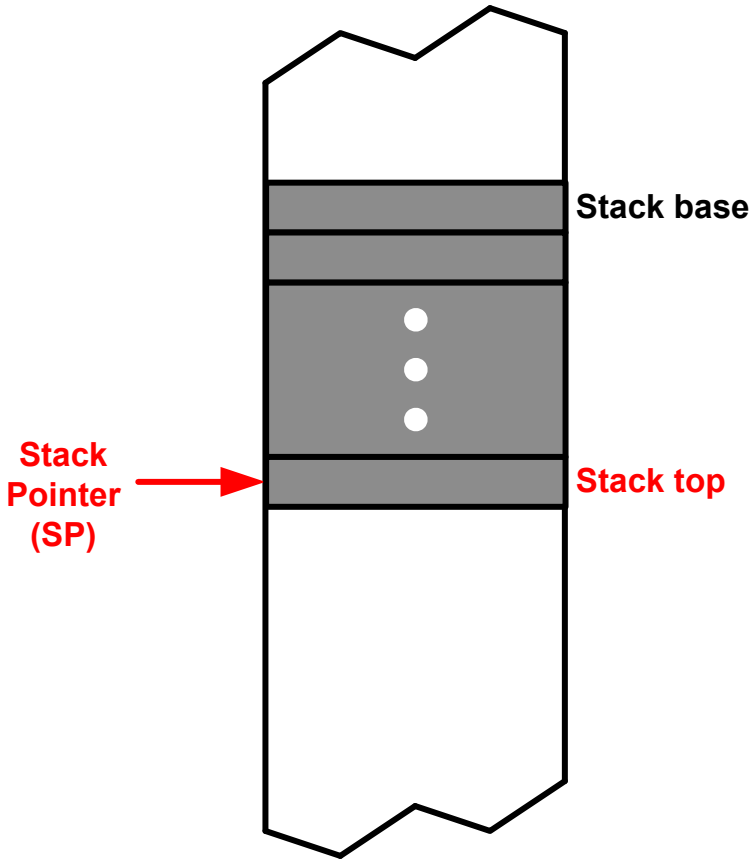


Descending stack: Stack grows towards low memory address

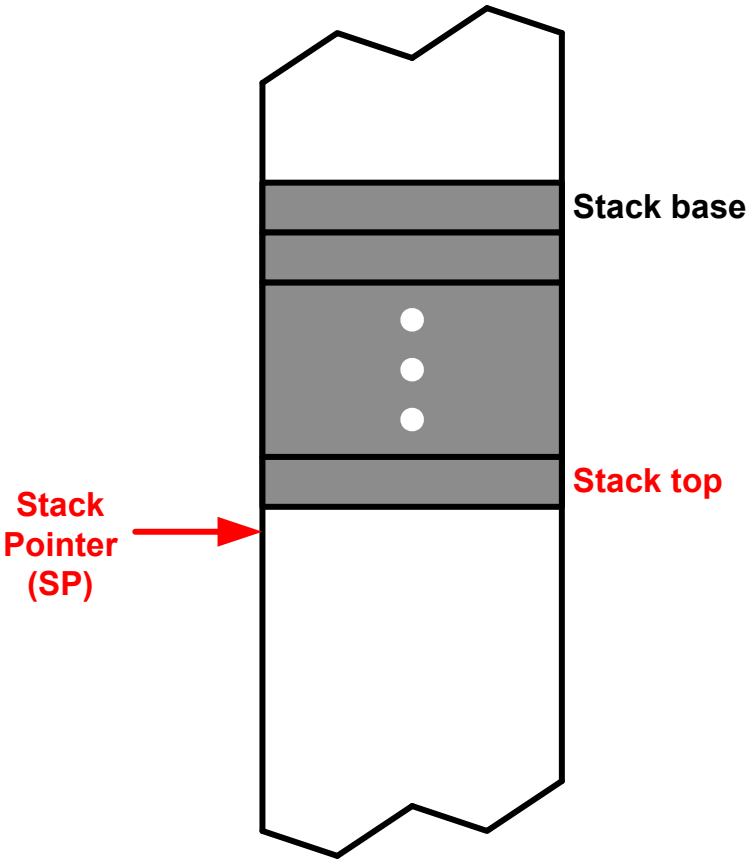


Ascending stack: Stack grows towards high memory address

Stack Growth Convention: Full *vs* Empty



Full stack: SP points to the last item pushed onto the stack



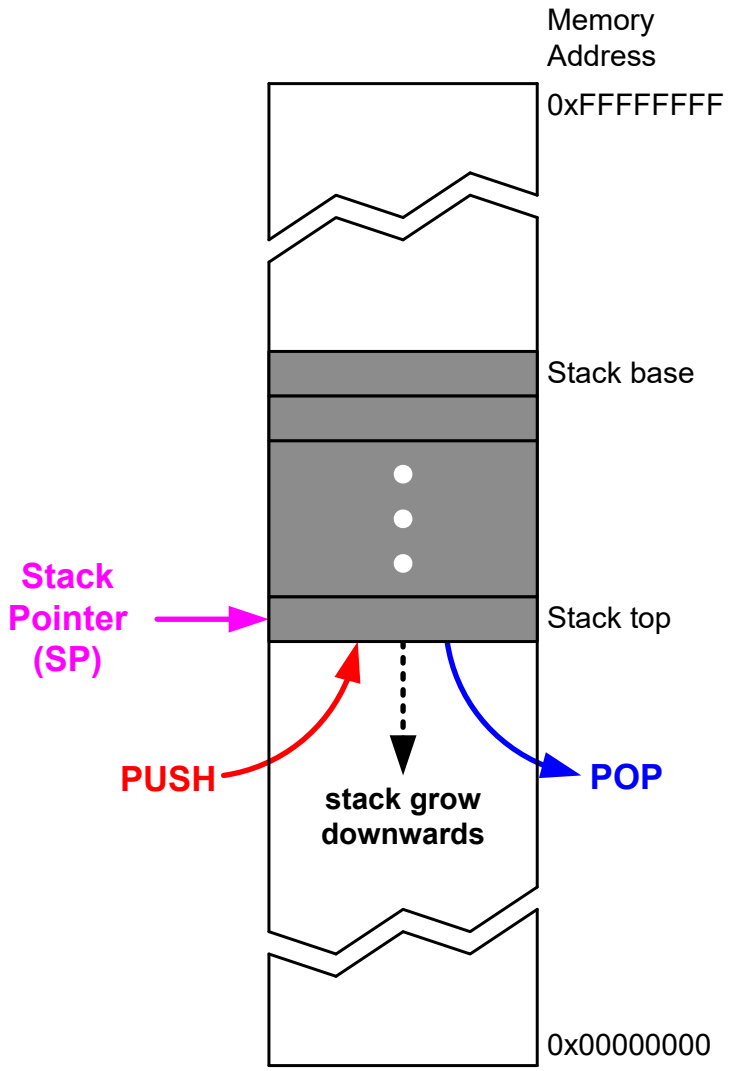
Empty stack: SP points to the next free space on the stack

Cortex-M Stack

- Cortex-M uses ***full descending stack***
- Example:

```
PUSH/POP {r0,r6,r3}
```

- Stack pointer (SP, aka R13)
  - decremented on **PUSH**
    - **SP = SP – 4 * # of registers**
  - incremented on **POP**
    - **SP = SP + 4 * # of registers**
  - SP starts at **0x20000200** for STM32-Discovery by default (can be changed in startup.s)



Review

Addressing Modes for Load/Store Multiple Registers

```
STMxx rn{!}, {register_list}
LDMxx rn{!}, {register_list}
```

- xx = IA, IB, DA, or DB. The order in which registers are listed does not matter
- For STM/LDM, the lowest-numbered register is stored/loaded at the lowest memory address.

Addressing Modes	Description	Instructions
IA	Increment After	STMIA, LDMIA
IB	Increment Before	STMIB, LDMIB
DA	Decrement After	STMDA, LDMDA
DB	Decrement Before	STMDB, LDMDB

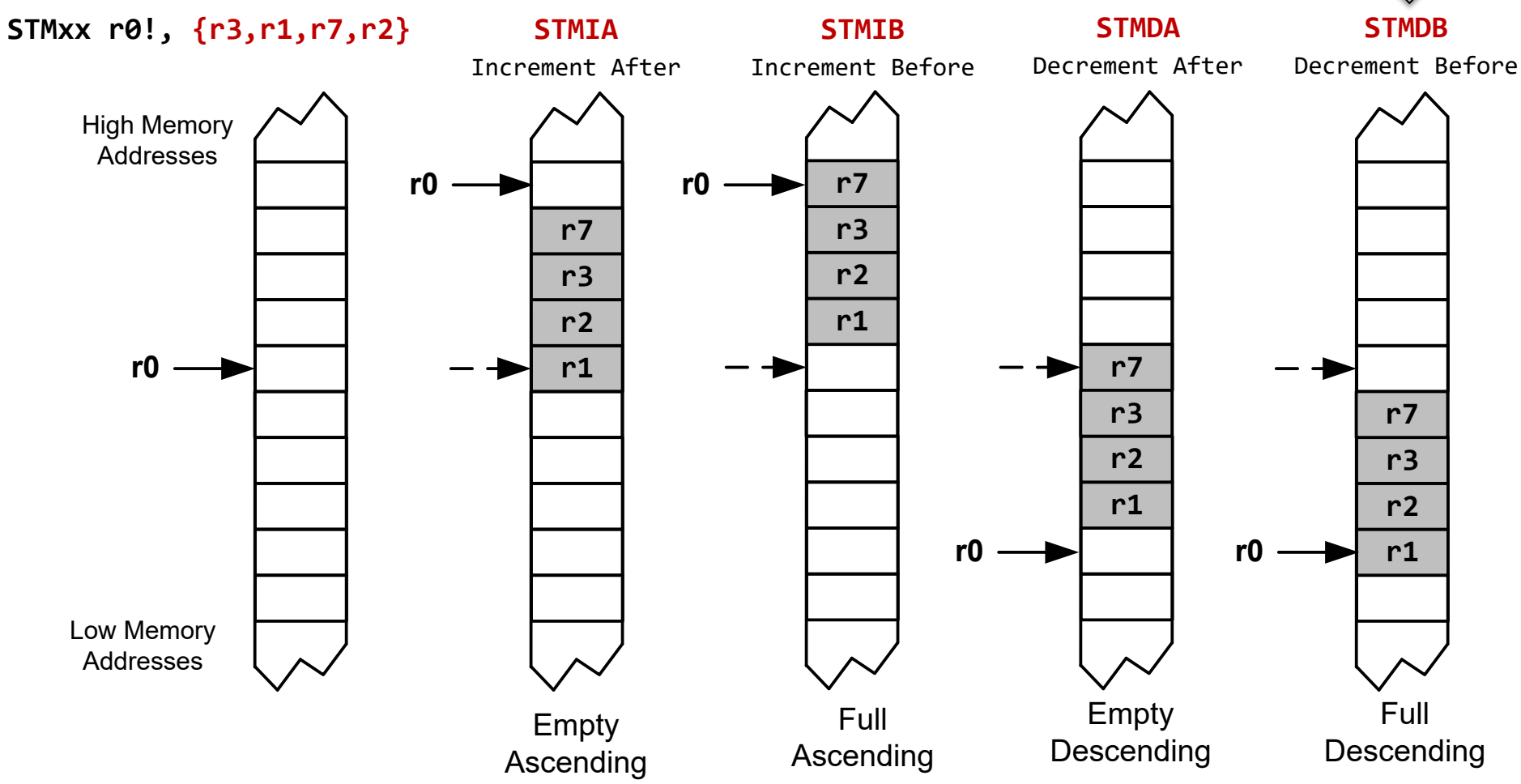
Cortex-M Stack 

- IA: address is incremented by 4 after a word is loaded or stored.
- IB: address is incremented by 4 before a word is loaded or stored.
- DA: address is decremented by 4 after a word is loaded or stored.
- DB: address is decremented by 4 before a word is loaded or stored.

Review

Store Multiple Registers

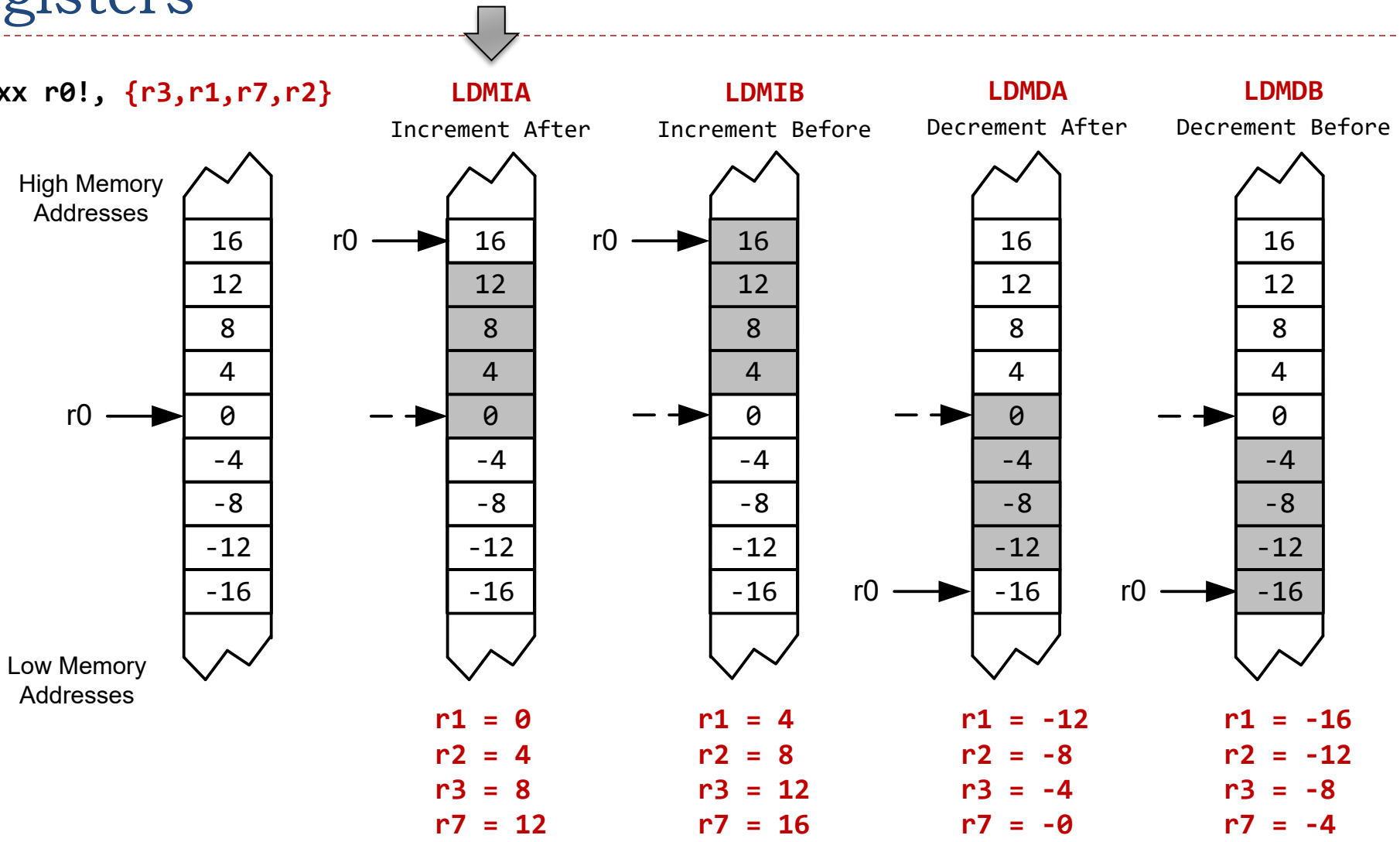
Cortex-M Stack



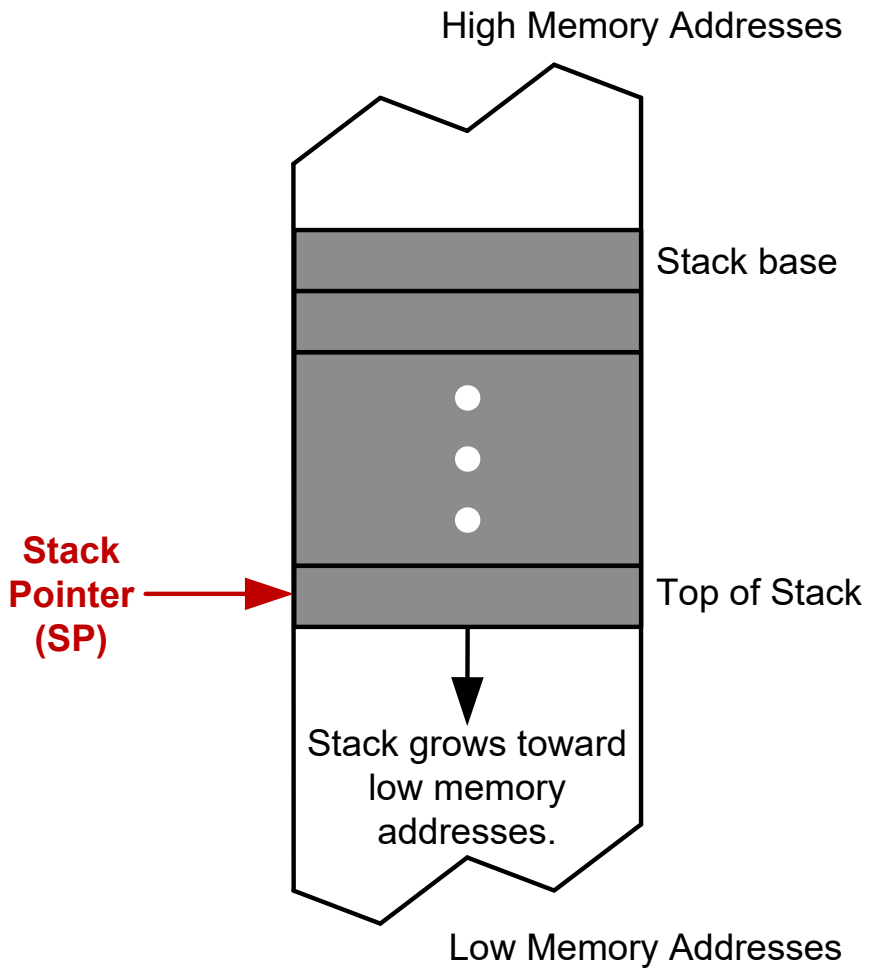
Review

Load Multiple Registers

Cortex-M Stack



Full Descending Stack



PUSH {register_list}
equivalent to:
STMDB SP!, {register_list}

DB: Decrement Before

POP {register_list}
equivalent to:
LDMIA SP!, {register_list}

IA: Increment After

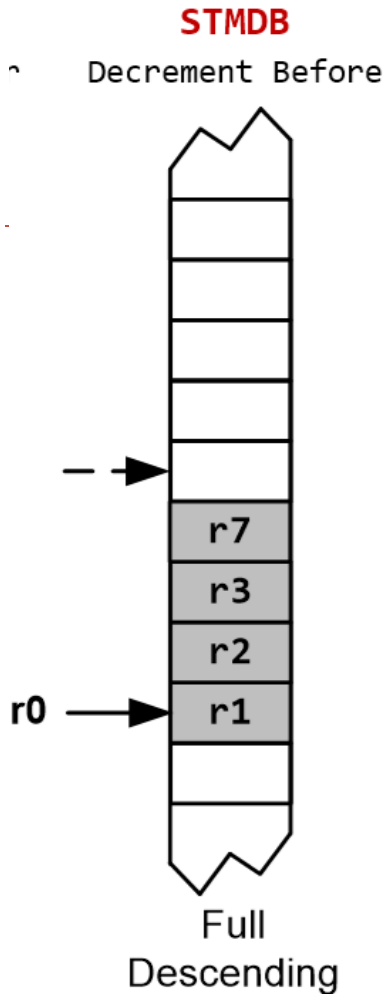
Stack Implementation (red text is Cortex-M stack)

Stock Name	Push		Pop	
	Equivalent	Alternative	Equivalent	Alternative
Full Descending(FD)	STMFD SP!,list	STMDB SP!,list	LDMFD SP!,list	LDMIA SP!,list
Empty Descending(ED)	STMED SP!,list	STMDA SP!,list	LDMED SP!,list	LDMIB SP!,list
Full Ascending(FA)	STMFA SP!,list	STMIB SP!,list	LDMFA SP!,list	LDMDA SP!,list
Empty Ascending(EA)	STMEA SP!,list	STMIA SP!,list	LDMEA SP!,list	LDMDB SP!,list

Stack

PUSH {Rd} == STMDB SP!, {Rd} == STMFD SP!, {Rd}

- SUB SP, SP, #4 @ SP = SP-4 (descending stack)
- STR Rd, [SP] @ (*SP) = Rd (full stack)



Push multiple registers

```
PUSH {r1, r2, r3, r7}   <=>   PUSH {r7, r2, r3, r1}   <=>   PUSH {r7}
                                                             PUSH {r3}
                                                             PUSH {r2}
                                                             PUSH {r1}
```

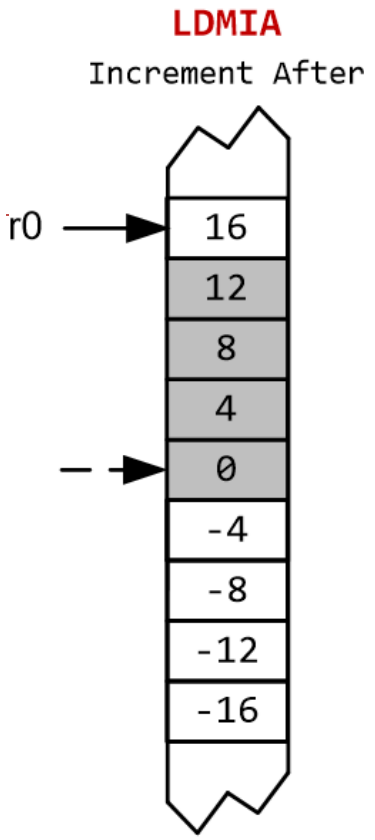
They are equivalent.

- SP is decremented before PUSH (pre-decrement).
- The order in which registers listed in the register list does not matter.
- When pushing multiple registers, these registers are automatically sorted by name and the lowest-numbered register is stored to the lowest memory address, *i.e.* r1 is stored last.

Stack

POP {*Rd*} == LDMIA SP!, {Rd} == LDMFD SP!, {Rd}

- LDR Rd, [SP] @ Rd = (*SP) (full stack)
- ADD SP, #4 @ SP = SP + 4 (Stack shrinks)



Pop multiple registers

They are equivalent.

```
POP {r1, r2, r3, r7}  ⟺  POP {r7, r3, r2, r1}  ⟺
POP {r1}
POP {r2}
POP {r3}
POP {r7}
```

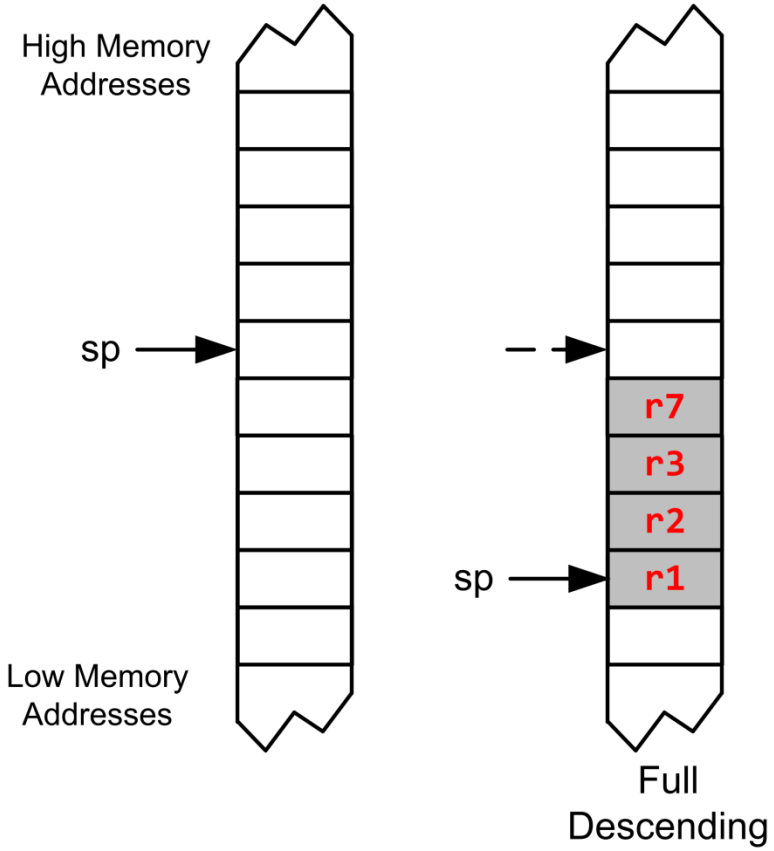
```
r1 = 0
r2 = 4
r3 = 8
r7 = 12
```

- SP is incremented after POP (post-increment).
- The order in which registers listed in the register list does not matter.
- When popping multiple registers, these registers are automatically sorted by name and the lowest-numbered register is loaded from the lowest memory address, *i.e.* r1 is loaded first.

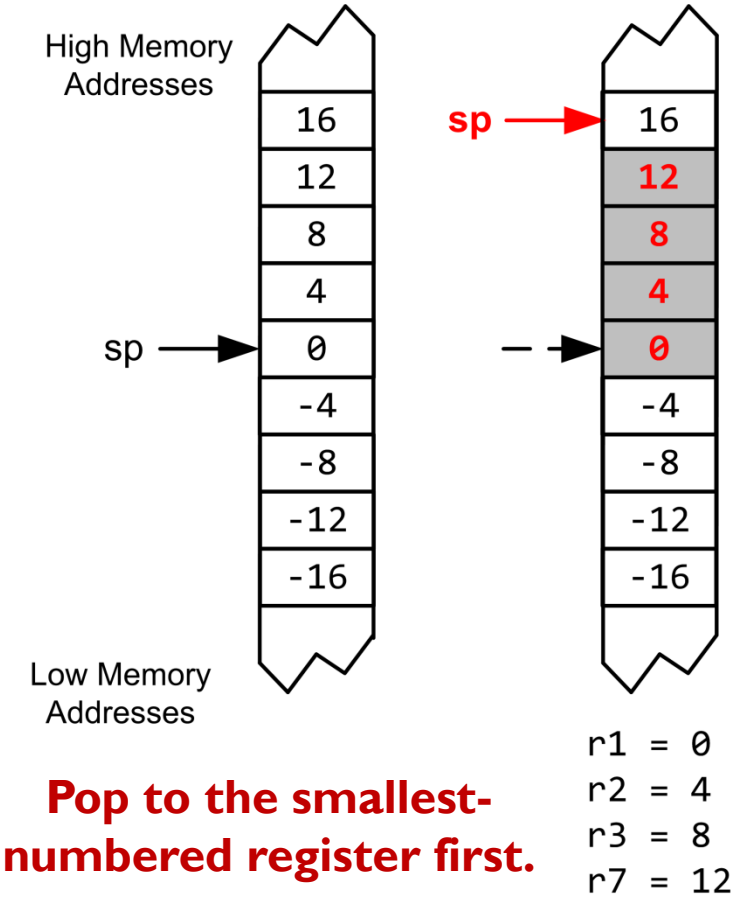
Stack Recap

Largest-numbered register is pushed first but popped last.

PUSH {r3, r1, r7, r2}



POP {r3, r1, r7, r2}



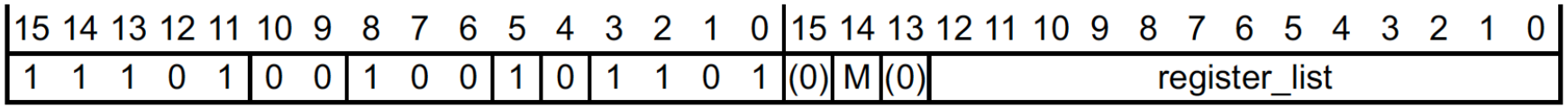
Pop to the smallest-numbered register first.

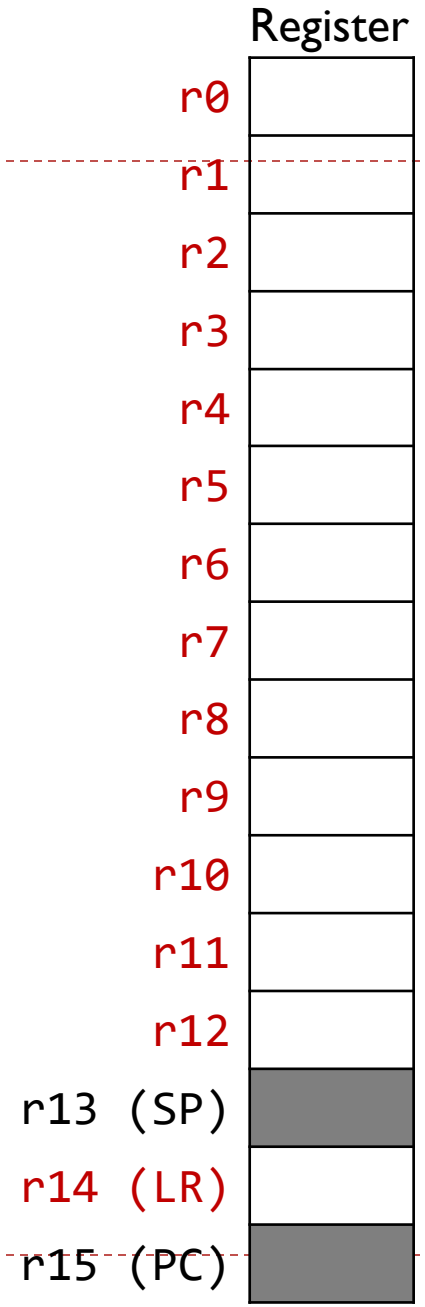
```
r1 = 0
r2 = 4
r3 = 8
r7 = 12
```

Cortex-M Stack

- `PUSH {register list}`
- `POP  {register list}`
  - Eligible registers for PUSH: `r0 - r12, LR`
  - Eligible registers for POP: `r0 - r12, LR, PC`
  - Order specified in the list does not matter
    - Largest-numbered register is always pushed first and popped last
    - Smallest-numbered register is always pushed last and popped first
    - Smallest-numbered register is stored at lowest memory address after PUSH

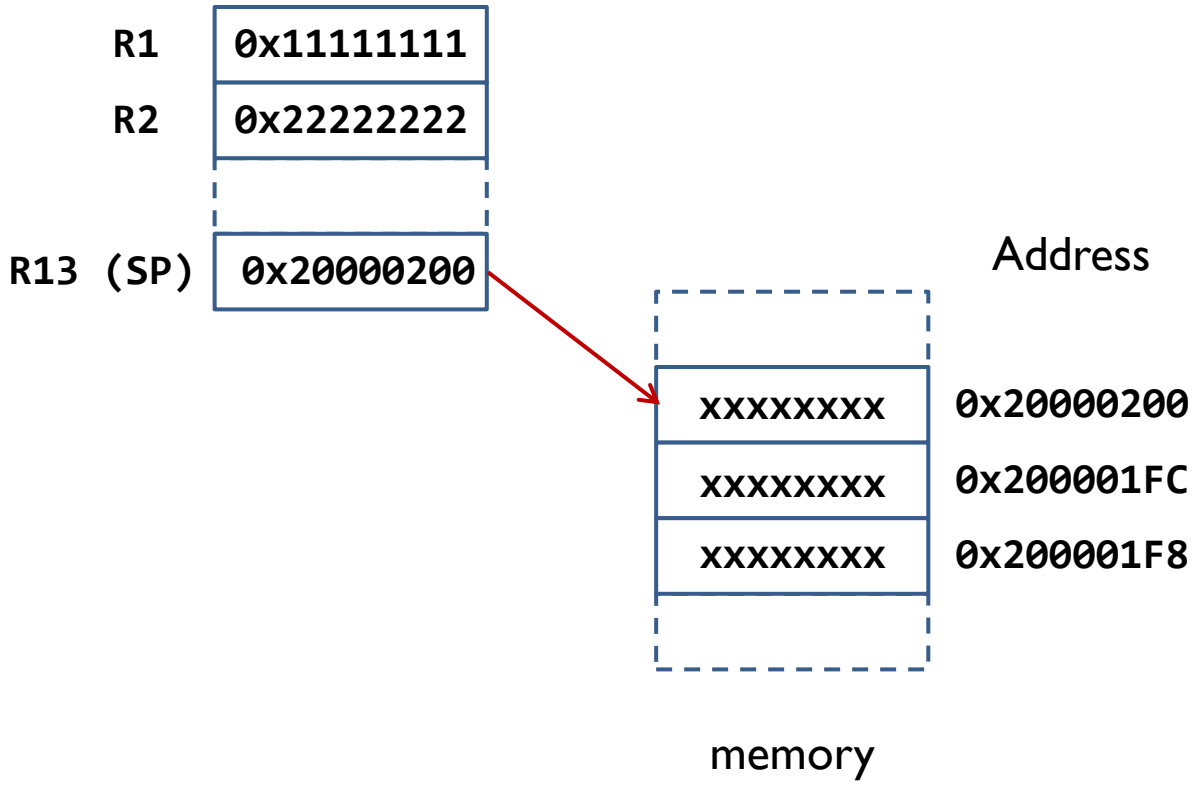
Instruction format for PUSH:



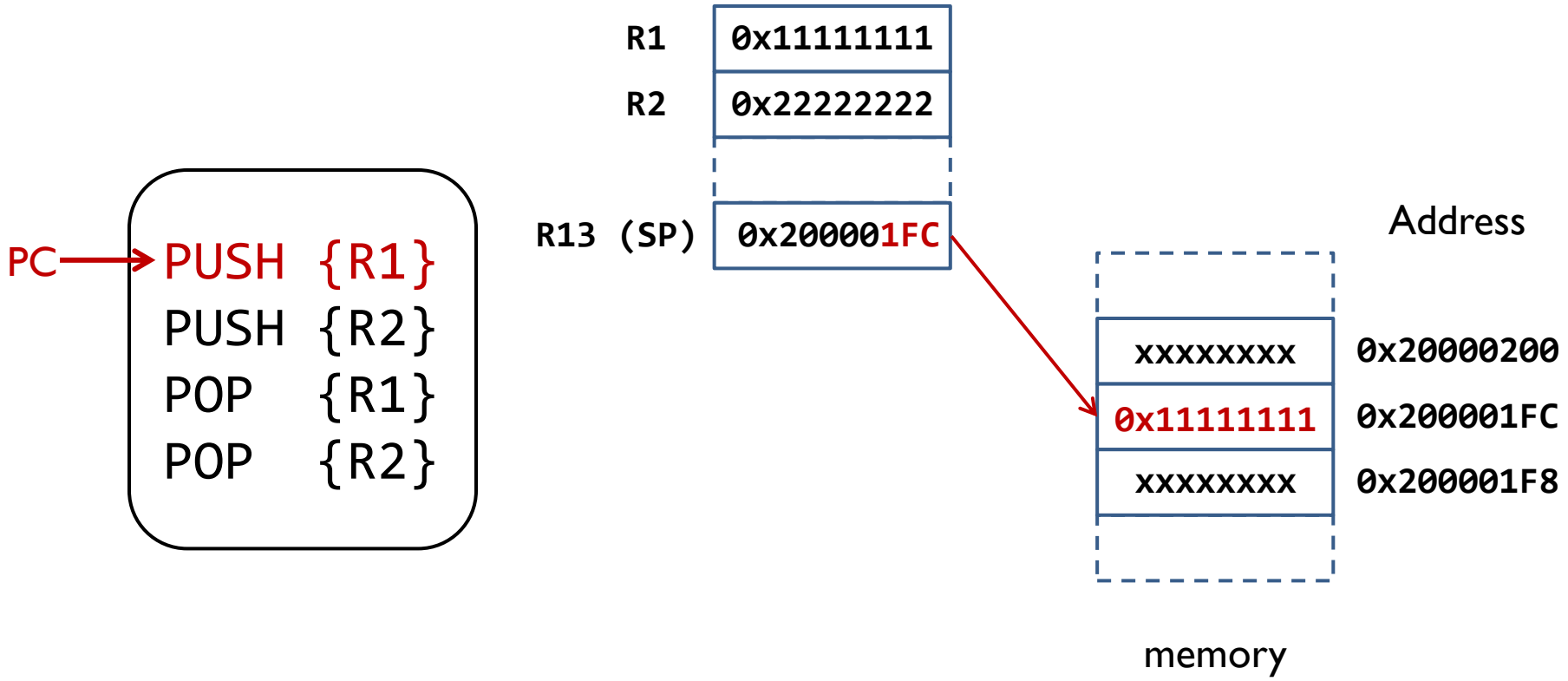


Example: swap R1 & R2

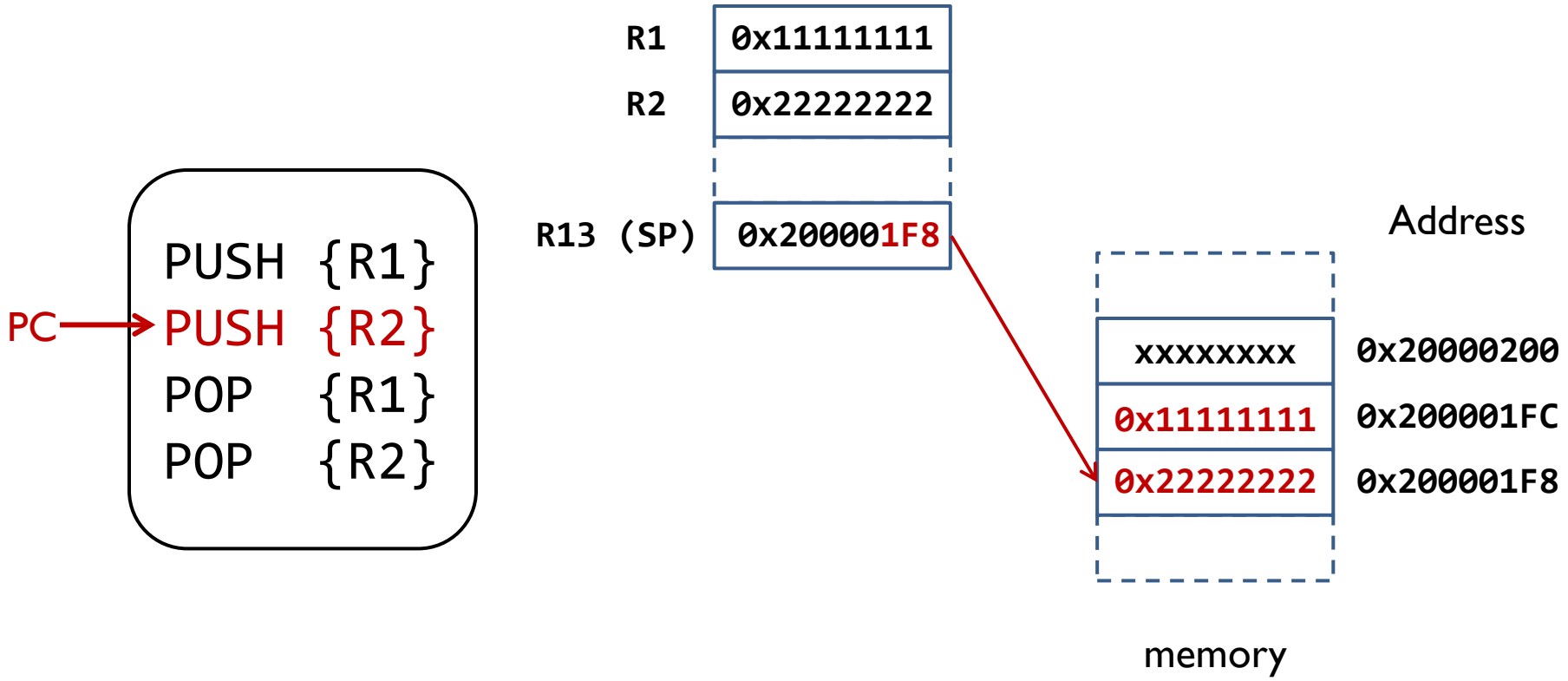
```
PUSH {R1}
PUSH {R2}
POP  {R1}
POP  {R2}
```



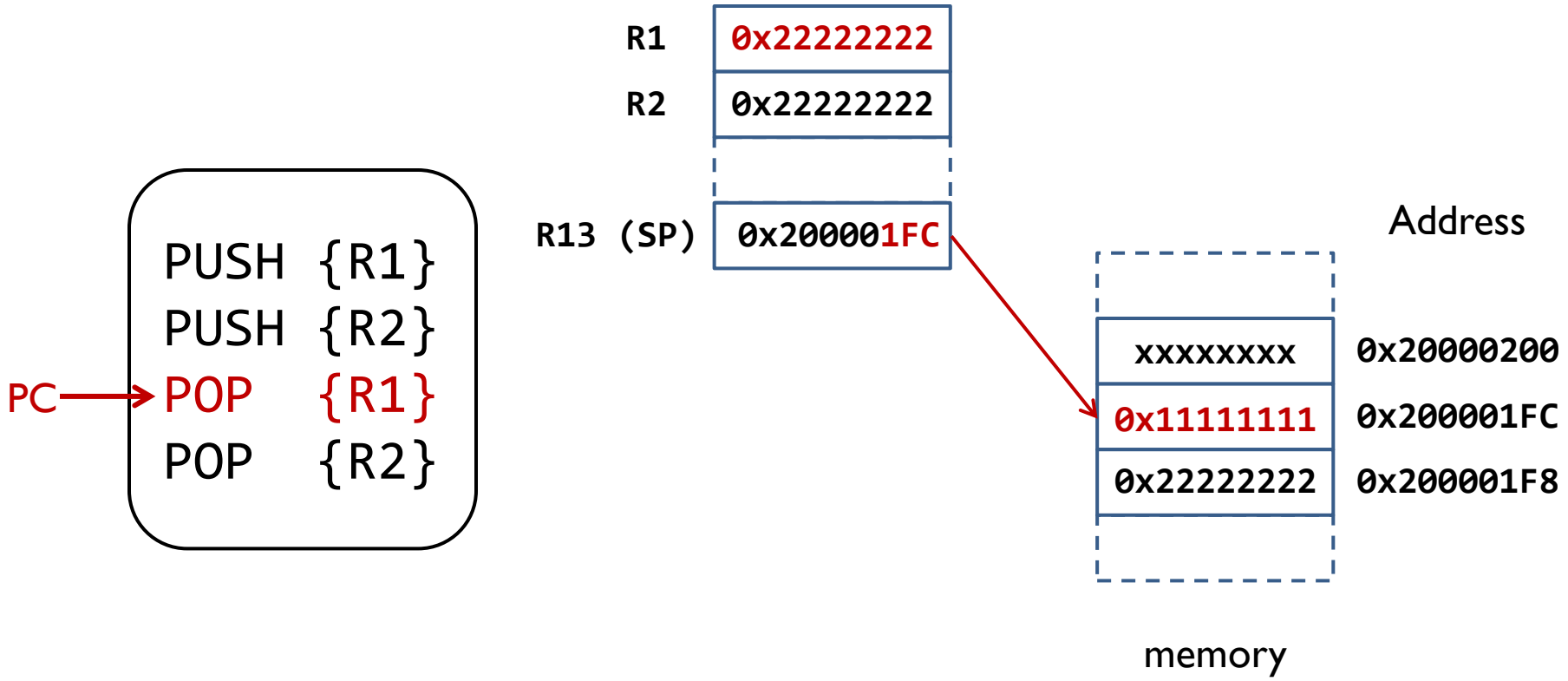
Example: swap R1 & R2



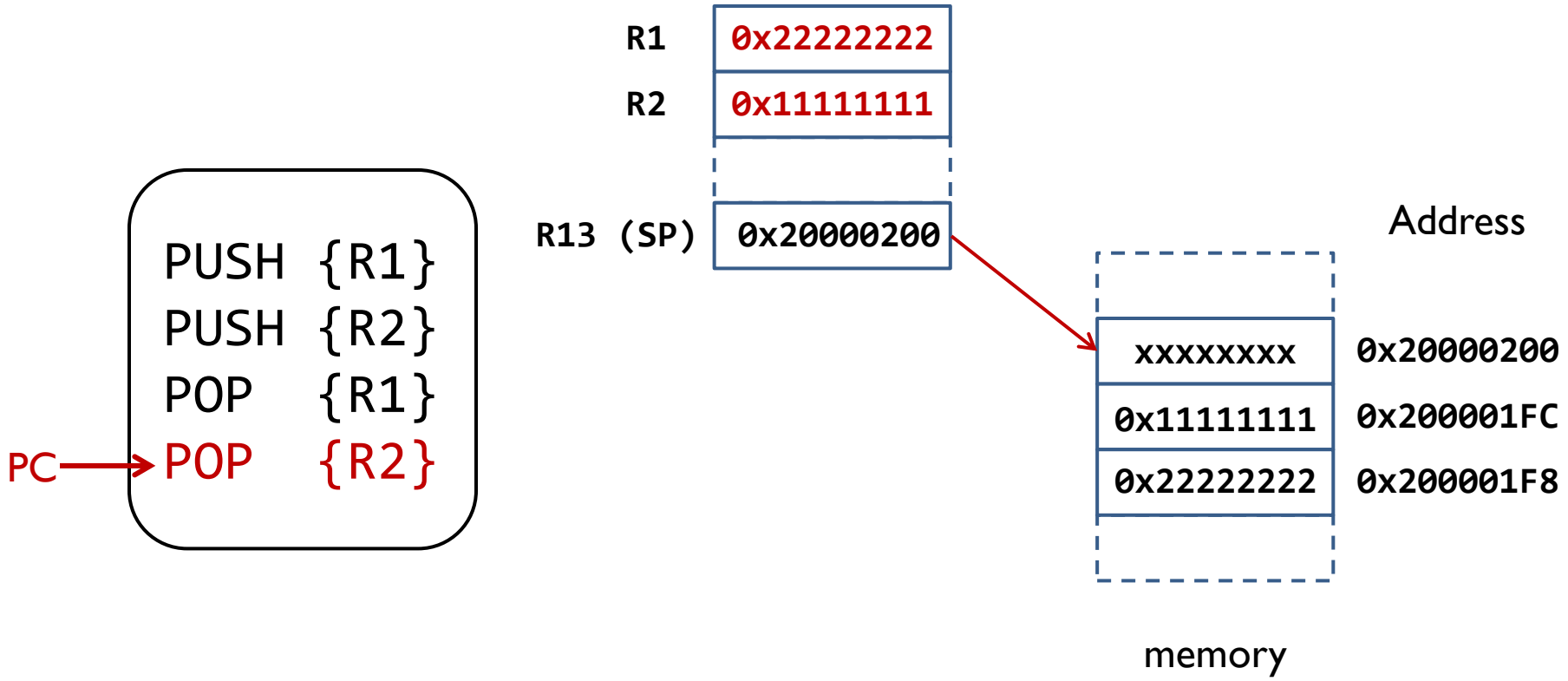
Example: swap R1 & R2



Example: swap R1 & R2



Example: swap R1 & R2



Values of R1 and R2 are swapped

Quiz

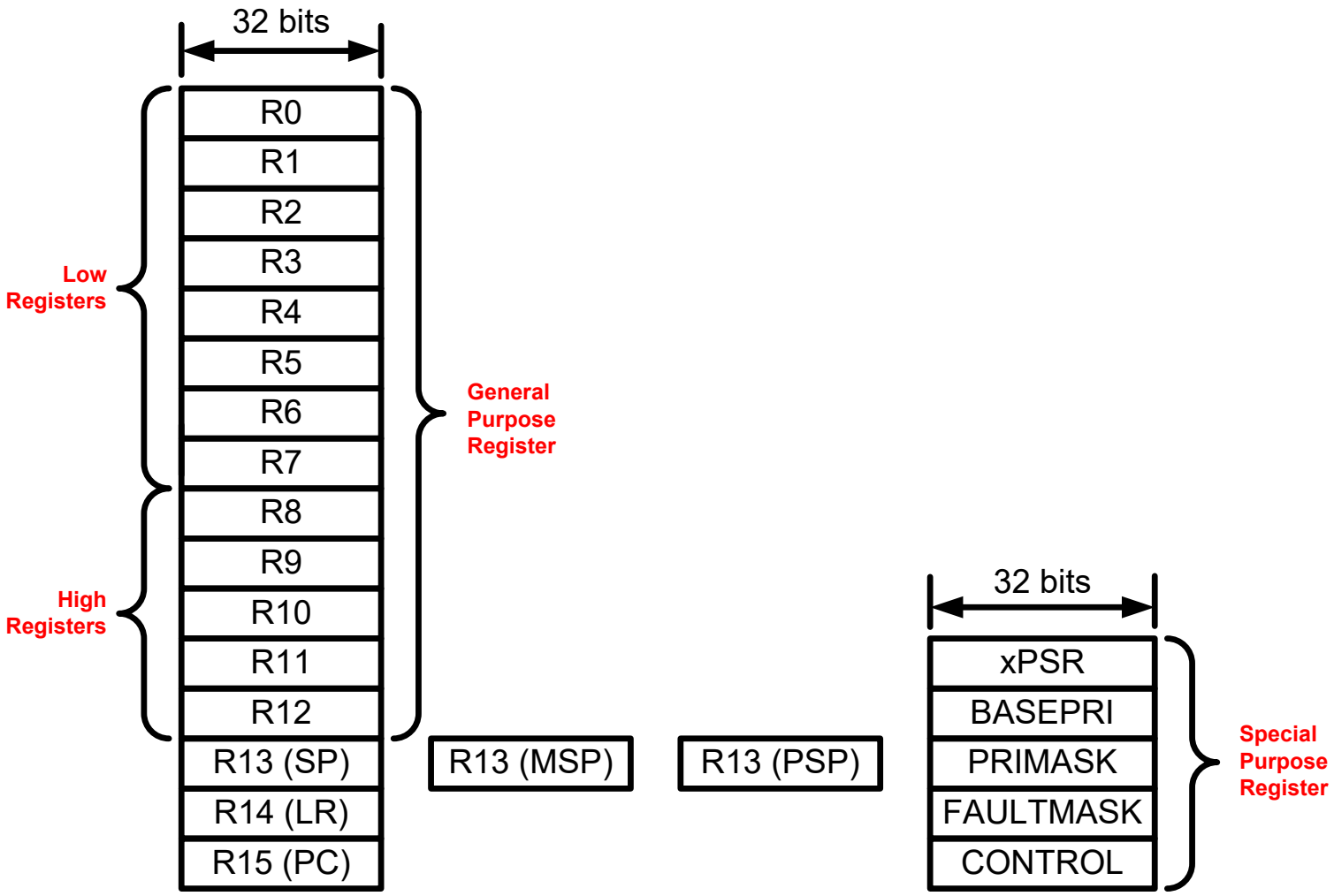
- Are the values of R1 and R2 swapped?
- PUSH {R1, R2}; POP {R2, R1}
- Or
- PUSH {R1, R2}; POP {R2}; POP {R1}
- Or
- PUSH {R1}; PUSH {R2}; POP {R1, R2}

Quiz ANS

- Are the values of R1 and R2 swapped? (not valid assembly syntax; need to put instructions on different lines, not using ; to separate them)
- PUSH {R1, R2}; POP {R2, R1}
- Or
- PUSH {R1, R2}; POP {R2}; POP {R1}
- Or
- PUSH {R1}; PUSH {R2}; POP {R1, R2}
- PUSH {R1, R2}; POP {R2, R1}
  - is equivalent to
  - PUSH {R2}; PUSH {R1}; POP{R1}; POP{R2}
  - Values of R1 and R2 are unchanged
- PUSH {R1, R2}; POP {R2}; POP {R1}
  - is equivalent to
  - PUSH {R2}; PUSH {R1}; POP{R2}; POP{R1}
  - Values of R1 and R2 are swapped
- PUSH {R1}; PUSH {R2}; POP {R1, R2}
  - is equivalent to
  - PUSH {R1}; PUSH {R2}; POP{R1}; POP{R2}
  - Values of R1 and R2 are swapped

Subroutine

- A subroutines, also called a function or a procedure,
  - single-entry, single-exit
  - Return to caller after it exits
- When a subroutine is called, the Link Register (LR) holds the return address of the current subroutine call, i.e., memory address of the next instruction to be executed after the subroutine exits.



Calling a Subroutine

Caller: **`BL`** ***label*** (Branch and Link)

- Step 1: `LR = PC + 4`
- Step 2: `PC = label`
  - *label* is name of subroutine
  - Compiler translates label to memory address
  - After call, LR holds return address (the instruction following the call)

Callee: **`BX LR`** (Branch and Exchange) at end of procedure

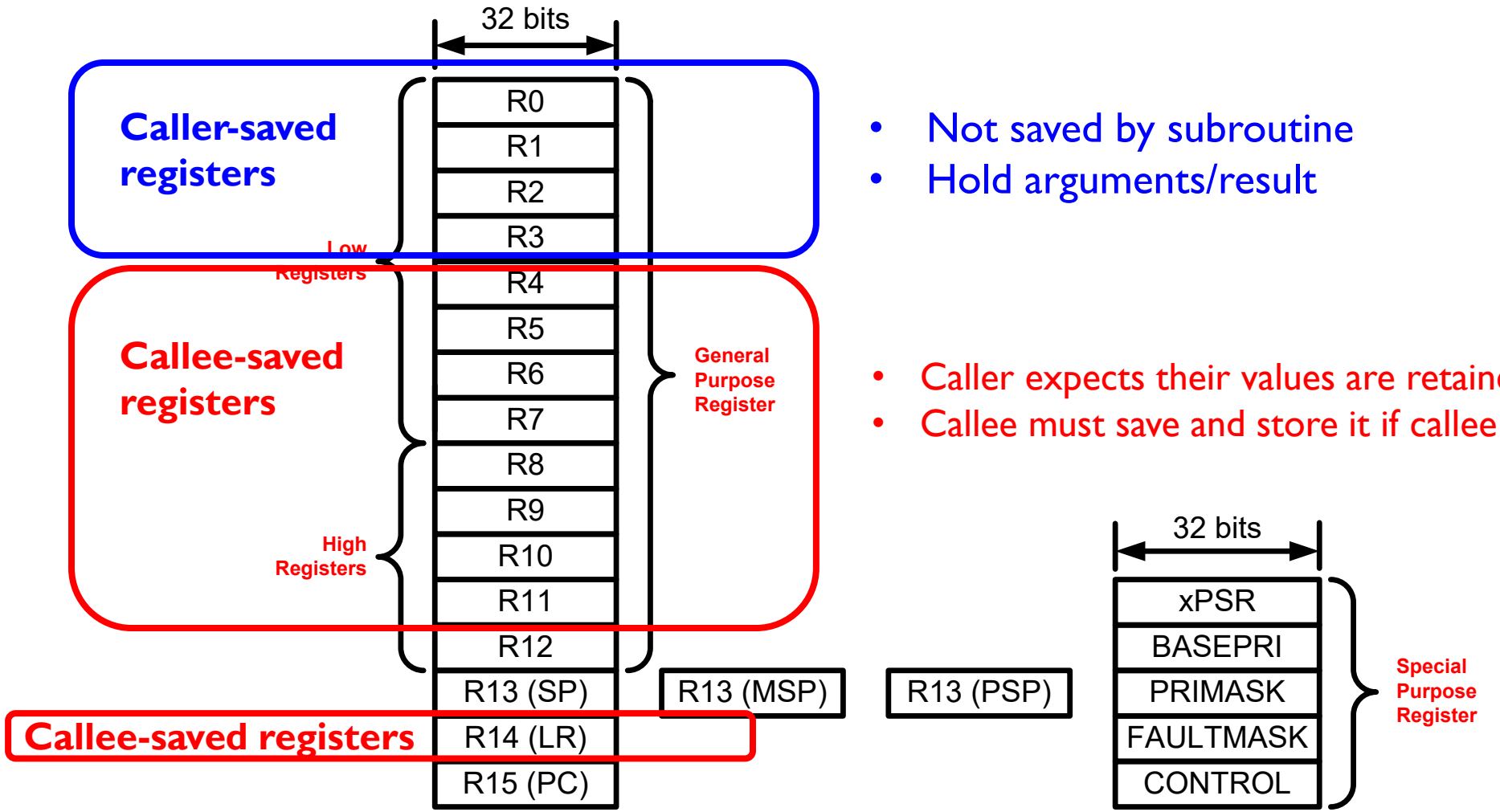
- `PC = LR`
  - Return to caller by setting PC to LR
- Equivalently:
  - PUSH {LR} at start of procedure
  - POP {PC} at end of procedure

Caller Program	Subroutine/Callee	
... **BL foo** ...	foo PROC ... **BX LR** **EDP**	foo PROC **PUSH {LR}** ... **POP {PC}** ; pops LR into PC (returns) **EDP**

ARM Procedure Call Standard

Register	Usage	Subroutine Preserved	Notes
r0	Argument 1 and return value	No	If return has 64 bits, then r0:r1 hold it. If argument 1 has 64 bits, r0:r1 hold it.
r1	Argument 2	No	
r2	Argument 3	No	If the return has 128 bits, r0-r3 hold it.
r3	Argument 4	No	If more than 4 arguments, use the stack
r4	General-purpose V1	Yes	Variable register 1 holds a local variable.
r5	General-purpose V2	Yes	Variable register 2 holds a local variable.
r6	General-purpose V3	Yes	Variable register 3 holds a local variable.
r7	General-purpose V4	Yes	Variable register 4 holds a local variable.
r8	General-purpose V5	YES	Variable register 5 holds a local variable.
r9	Platform specific/V6	Yes/No	Usage is platform-dependent.
r10	General-purpose V7	Yes	Variable register 7 holds a local variable.
r11	General-purpose V8	Yes	Variable register 8 holds a local variable.
r12 (IP)	Intra-procedure-call register	No	It holds intermediate values between a procedure and the sub-procedure it calls.
r13 (SP)	Stack pointer	Yes	SP has to be the same after a subroutine has completed.
r14 (LR)	Link register	No	LR does not have to contain the same value after a subroutine has completed.
r15 (PC)	Program counter	N/A	Do not directly change PC

Caller-saved Registers *vs* Callee-saved Registers



- Not saved by subroutine
- Hold arguments/result

- Caller expects their values are retained
- Callee must save and store it if callee modifies it

Embedded Application Binary Interface (EABI) Protocol

- Caller-saved registers:
  - R0–R3: Arguments/return registers.
  - R12 (IP): Intra-procedure scratch register.
  - CPSR: — caller must preserve its state if needed.
- Callee-saved registers:
  - R4–R11: Must be saved/restored by the callee if used.
  - R14 (LR): Must be saved if the callee makes nested calls.
  - R13 (SP): Stack pointer — must be preserved.
- Extra parameters passed on stack:
  - When more than four arguments exist.
- Return value in R0

Role	Must Preserve	Notes
Caller	R0–R3, R12, CPSR (if needed)	Caller-saved (scratch)
Callee	R4–R11, R13 (SP), R14 (LR)	Must be saved/restored if modified
Return	R0	Return value

- Top code (not good):
  - Callee foo does MOV R4, #10 → this overwrites whatever the caller had in R4.
  - foo does BX LR → returns to caller, and R4=10 after call return.
- Middle code:
  - Caller expects callee to not modify r4.
  - Callee should preserve r4 by saving and restores R4 by PUSH and POP on the stack
- Bottom code:
  - Callee saves and restores R4 by PUSH and POP
  - Callee uses PUSH{LR} and POP{PC}, equivalent to BX LR

Caller Program	Subroutine/Callee

```
MOV r4, #100
...
BL  foo
...
ADD r4, r4, #1    ; r4 = 11
```

```
foo PROC
    ...
    MOV   r4, #10   ; foo changes r4
    ...
    BX    LR
EDP
```

Caller Program	Subroutine/Callee

```
MOV r4, #100
...
BL  foo
...
ADD r4, r4, #1    ; r4 = 101
```

```
foo PROC
    PUSH {r4}      ; save caller's R4
    MOV  r4, #10
    POP  {r4}  ; restore caller's R4
    BX   LR
EDP
```

Caller Program	Subroutine/Callee

```
MOV r4, #100
...
BL  foo
...
ADD r4, r4, #1    ; r4 = 101
```

```
foo PROC
    PUSH {r4, LR} ; pushes LR before
r4
    MOV  r4, #10
    POP   {r4, PC} ; pops R4 before
LR, so now PC = LR (returns)
EDP
```

Example Program

- Caller is responsible for preserving R1.
- Callee func1 is responsible for preserving R4 and LR.
- Func1 is both a callee and a caller. Here func1 does not need to preserve R0, since:
  - We overwrite R0 with "MOV R0, #2" before calling func2. We then rely on the return value in R0 after func2 updates R0, and we do not need the old R0 anymore.
- If func1 had needed the original value of R0 (say, it wanted to use both the original and the returned value), then it would need to preserve it by PUSH and POP before and after "BL func2"

```
__main PROC
       MOV     R1, #0
       MOV     R4, #1
       PUSH    {R1}
       BL      func1
       POP     {R1}
       ADD     R2, R1, R4
       ADD     R3, R2, R0
loop   B       loop
       ENDP

func1  PROC
       PUSH    {R4, LR}
       MOV     R0, #2
       BL      func2
       MOV     R4, R0
       ADD     R1, R0, R4
       POP     {R4, LR}
       BX      LR
       ENDP

func2  PROC
       ADD     R0, R0, #1
       BX      LR
       ENDP
```

Nested Subroutines: What is wrong?

Caller Program	Subroutine foo	Subroutine bar

MOV r4, #100
...
BL foo
...
ADD r4, r4, #1
	foo PROC
	PUSH {r4}
	...
	MOV r4, #10
	...
	BL bar
	...
	POP {r4}
	BX LR
	ENDP
		bar PROC
		...
		BX LR
		ENDP

- Caller (main function) does BL foo → LR = return address back into the caller.
- foo does PUSH {r4} (saves r4) but does not save LR.
- foo does BL bar → this instruction overwrites LR with the return address back into foo (i.e. the instruction after BL bar).
- bar returns (BX LR) into foo (normal), but the original LR that pointed back to the caller was lost.
- foo does POP {r4} then BX LR — but LR now points to the instruction inside foo (not to the caller), so foo does not return to the caller (main function)

Nested Subroutines: Solution #1

foo saves and restores its LR for returning to its caller, before calling bar.
(Without saving and restoring LR in foo, "BX LR" in foo will jump to instruction after "BL bar" in foo, and program is stuck in an infinite loop within foo.)

Caller Program	Subroutine foo	Subroutine bar

MOV r4, #100
...
BL foo
...
ADD r4, r4, #1
	foo PROC
	PUSH {r4, LR}
	...
	MOV r4, #10
	...
	BL bar
	...
	POP {r4, LR}
	BX LR
	ENDP
		bar PROC
		...
		BX LR
		ENDP

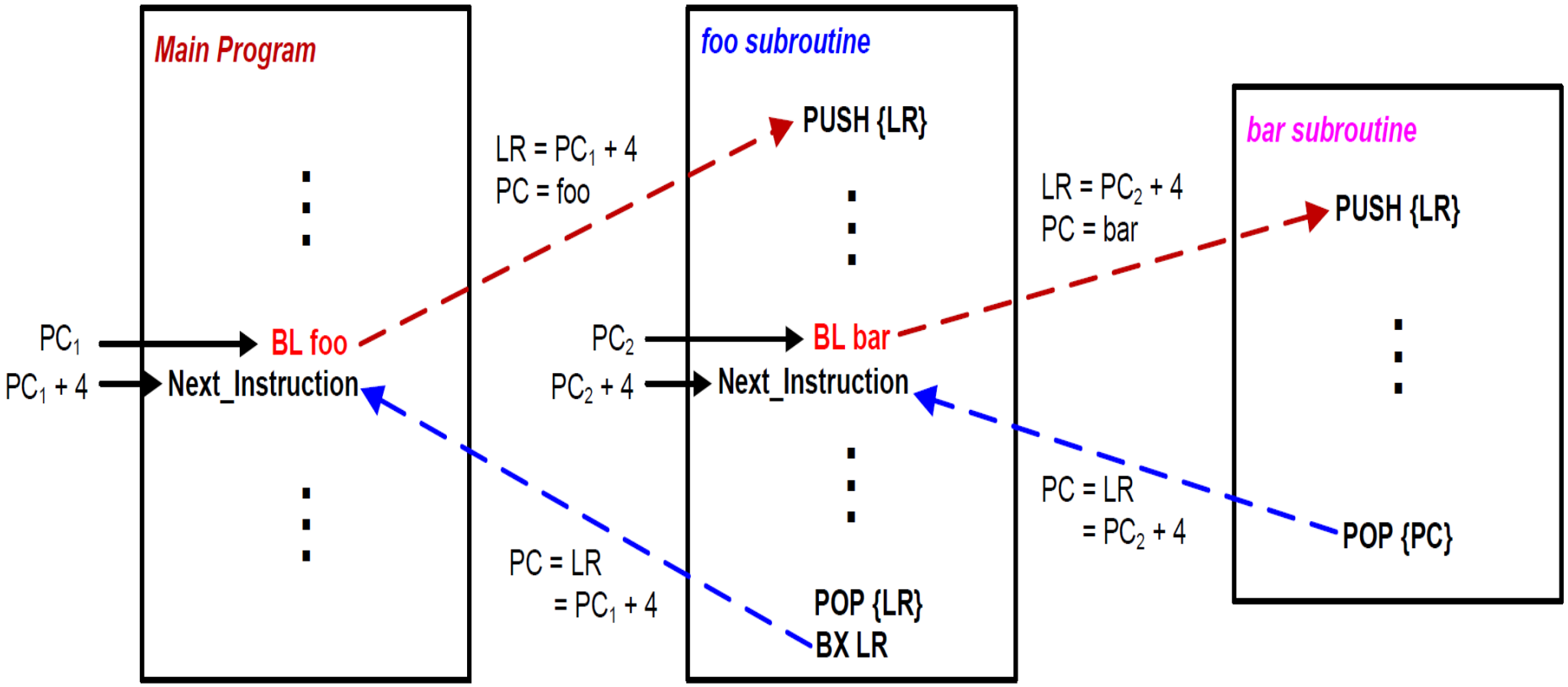
Nested Subroutines: Solution #2

POP {r4, PC} is equivalent to POP {r4, LR} followed by BX LR.

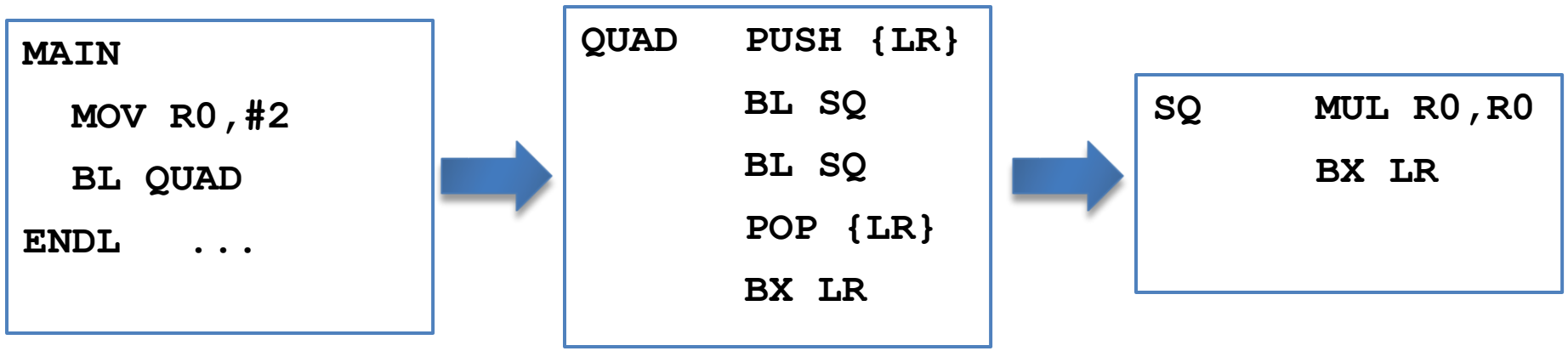
Caller Program	Subroutine foo	Subroutine bar

MOV r4, #100
...
BL foo
...
ADD r4, r4, #1
	foo PROC
	PUSH {r4, LR}
	...
	MOV r4, #10
	...
	BL bar
	...
	POP {r4, PC}
	~~BX LR~~
	ENDP
		bar PROC
		...
		BX LR
		ENDP

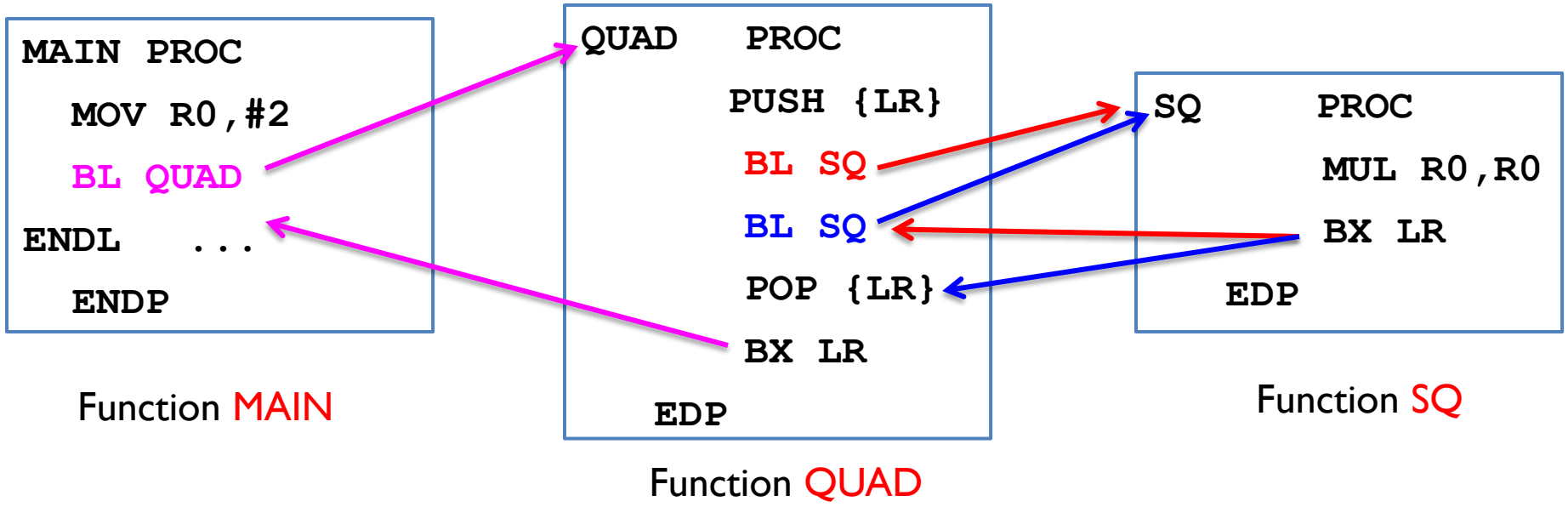
Nested Subroutines: Solution #1



Subroutine Calling Another Subroutine



Subroutine Calling Another Subroutine

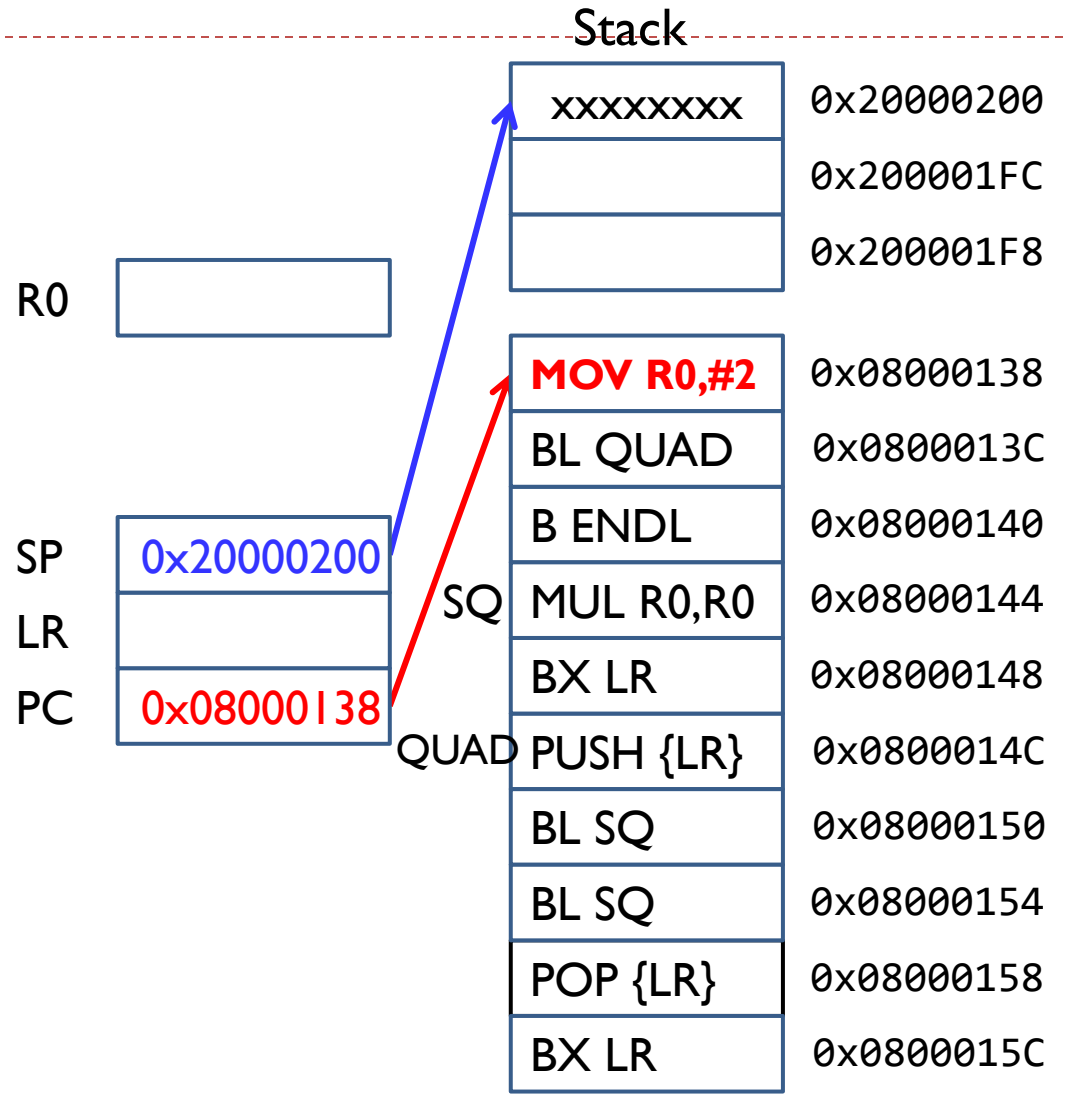


Example: R0 = R0⁴

```
        MOV R0,#2
        BL QUAD
        B ENDL

SQ      MUL R0,R0
        BX LR

QUAD    PUSH {LR}
        BL SQ
        BL SQ
        POP {LR}
        BX LR
ENDL    ...
```

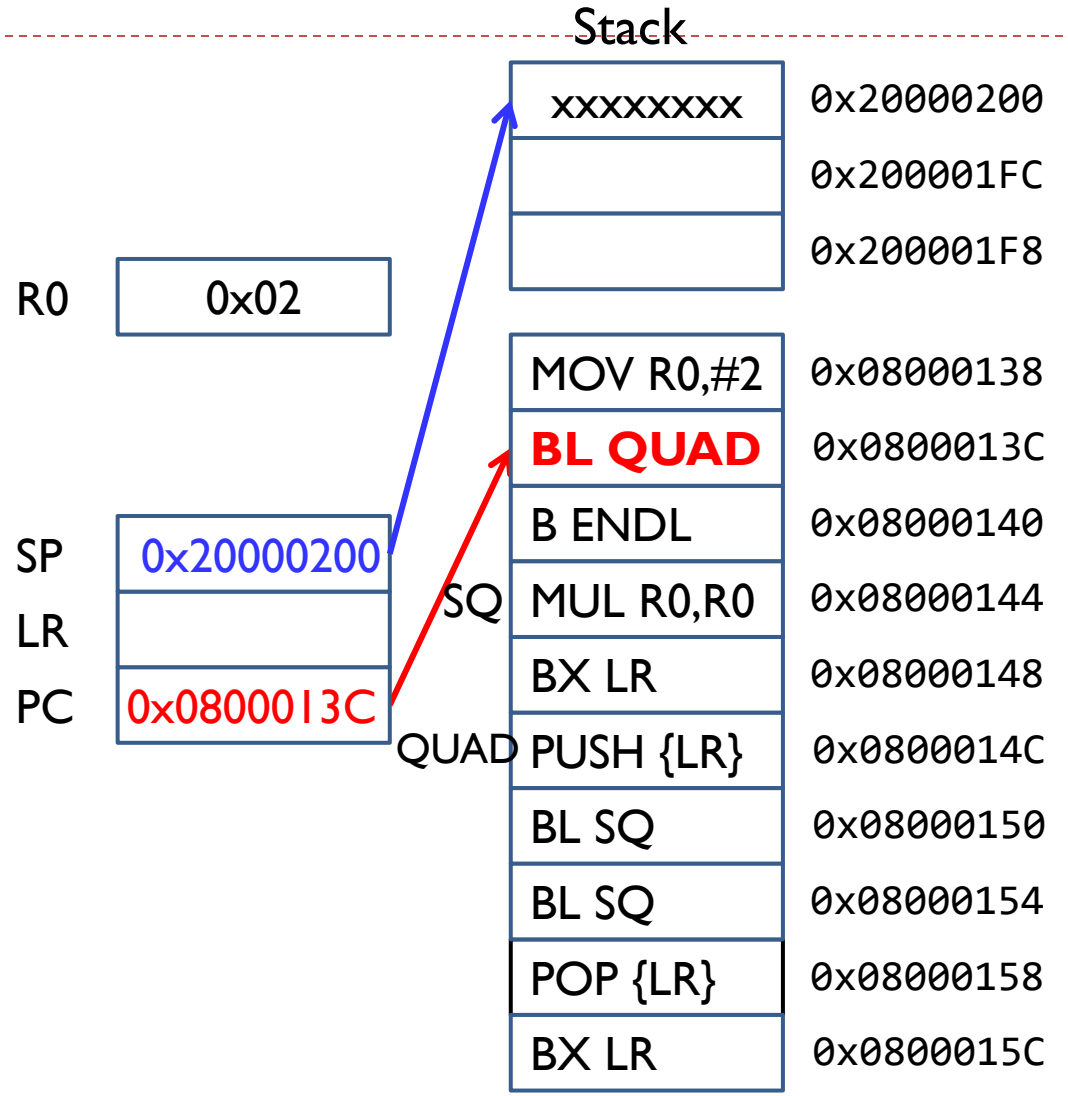


Example: R0 = R0⁴

```
        MOV R0,#2
        BL QUAD
        B ENDL

SQ      MUL R0,R0
        BX LR

QUAD    PUSH {LR}
        BL SQ
        BL SQ
        POP {LR}
        BX LR
ENDL    ...
```

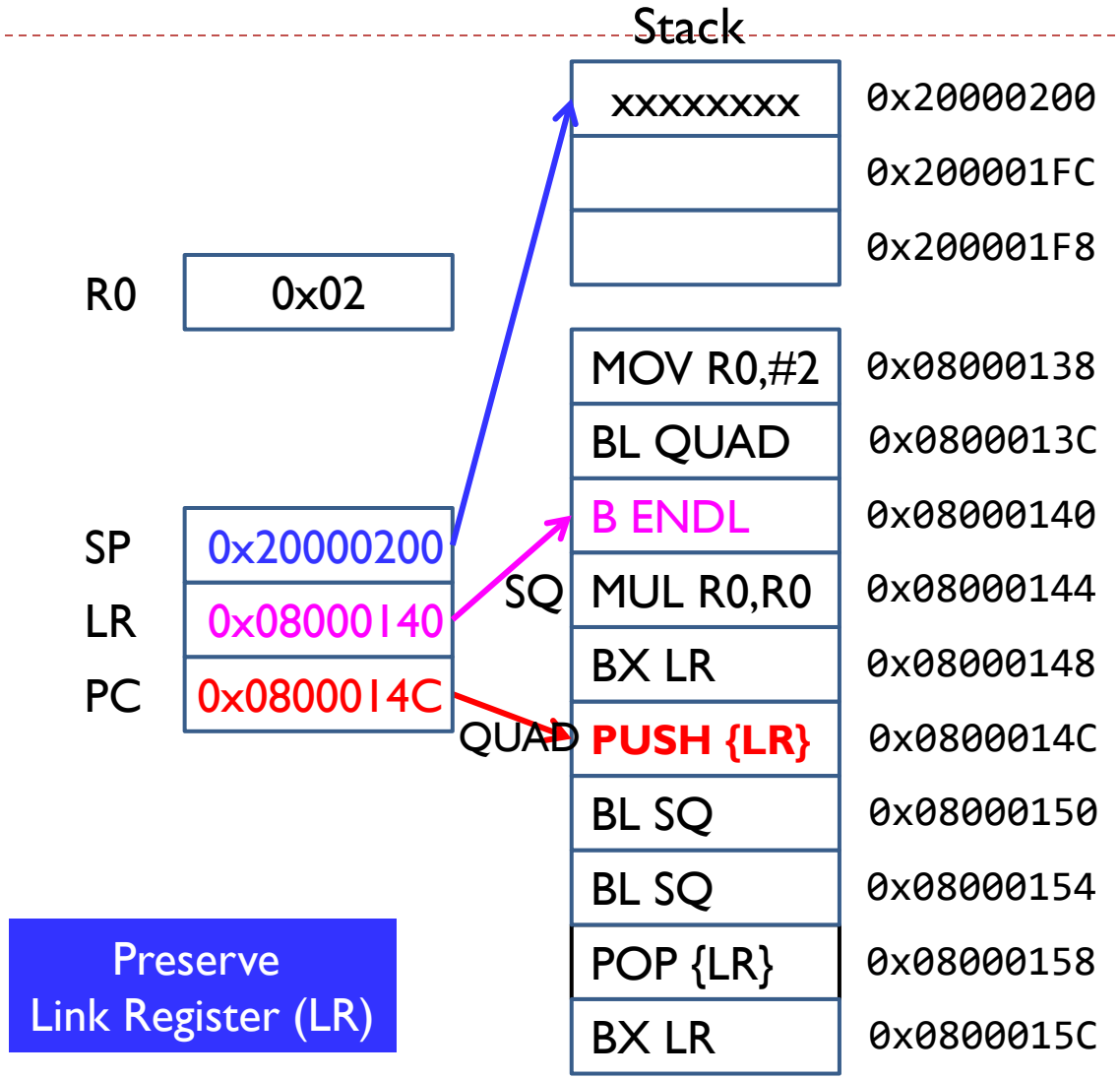


Example: R0 = R0^4

```
        MOV R0,#2
        BL QUAD
        B ENDL

SQ      MUL R0,R0
        BX LR

QUAD    PUSH {LR}
        BL SQ
        BL SQ
        POP {LR}
        BX LR
ENDL    ...
```

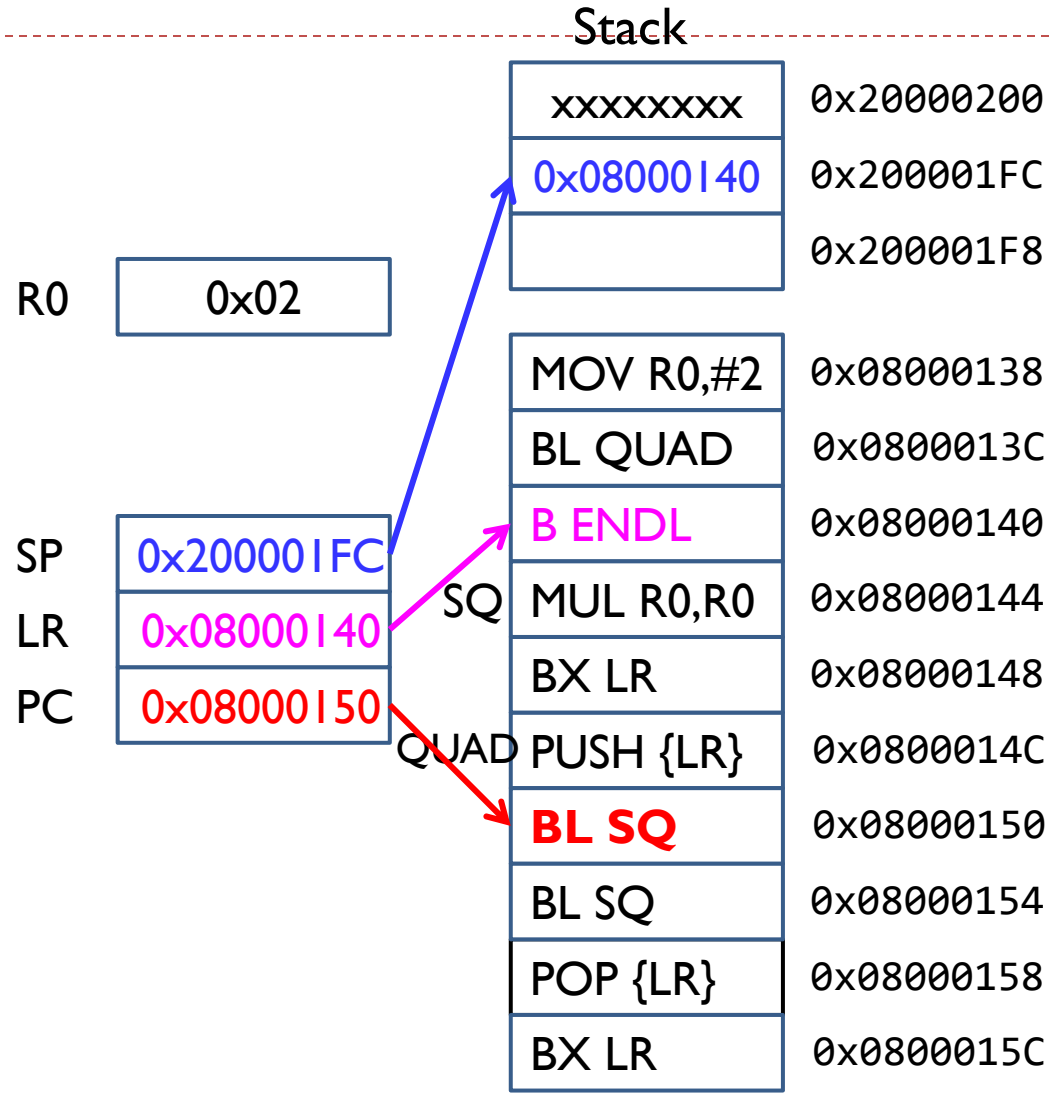


Example: R0 = R0^4

```
        MOV R0,#2
        BL QUAD
        B ENDL

SQ      MUL R0,R0
        BX LR

QUAD    PUSH {LR}
        BL SQ
        BL SQ
        POP {LR}
        BX LR
ENDL    ...
```

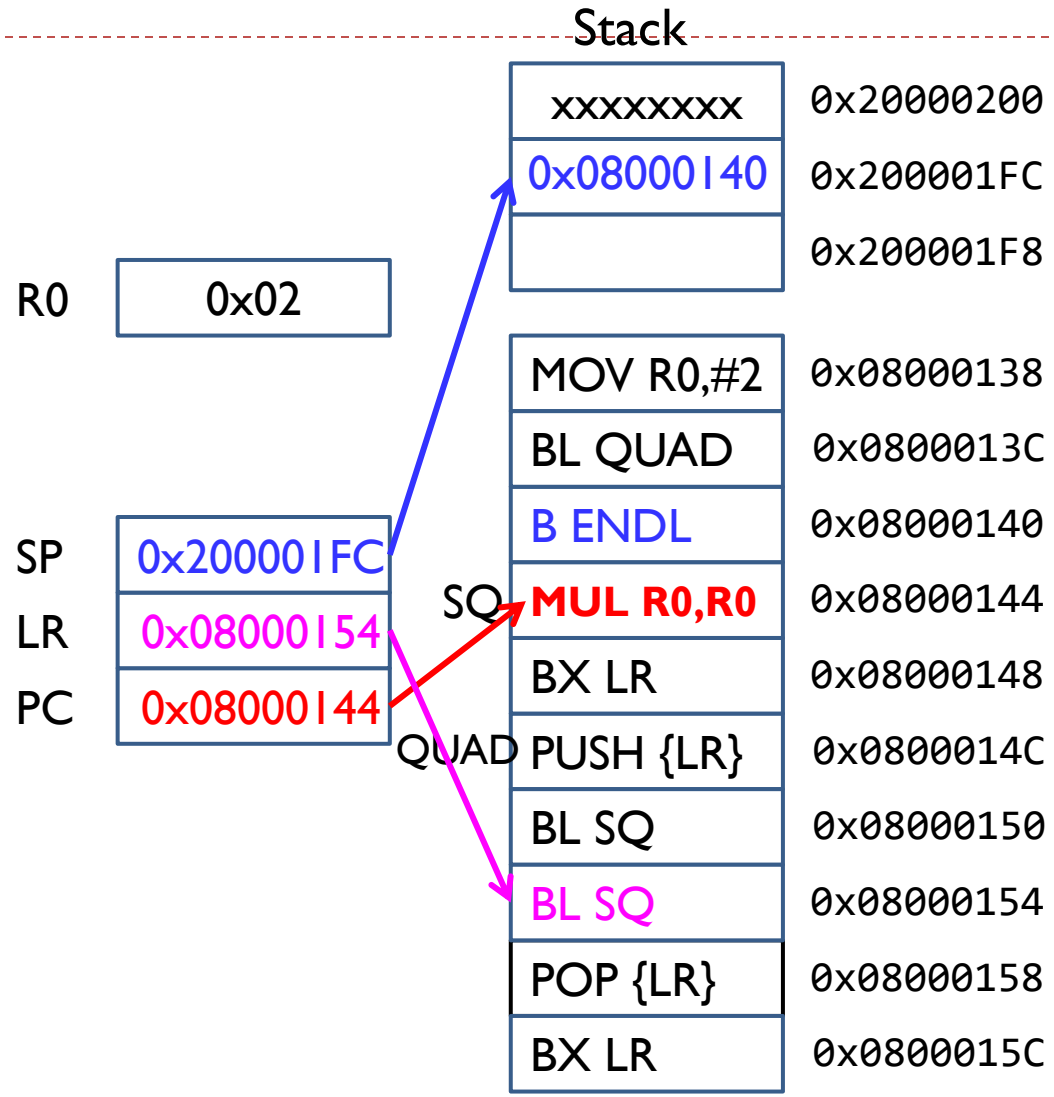


Example: R0 = R0^4

```
        MOV R0,#2
        BL QUAD
        B ENDL

SQ      MUL R0,R0
        BX LR

QUAD    PUSH {LR}
        BL SQ
        BL SQ
        POP {LR}
        BX LR
ENDL    ...
```

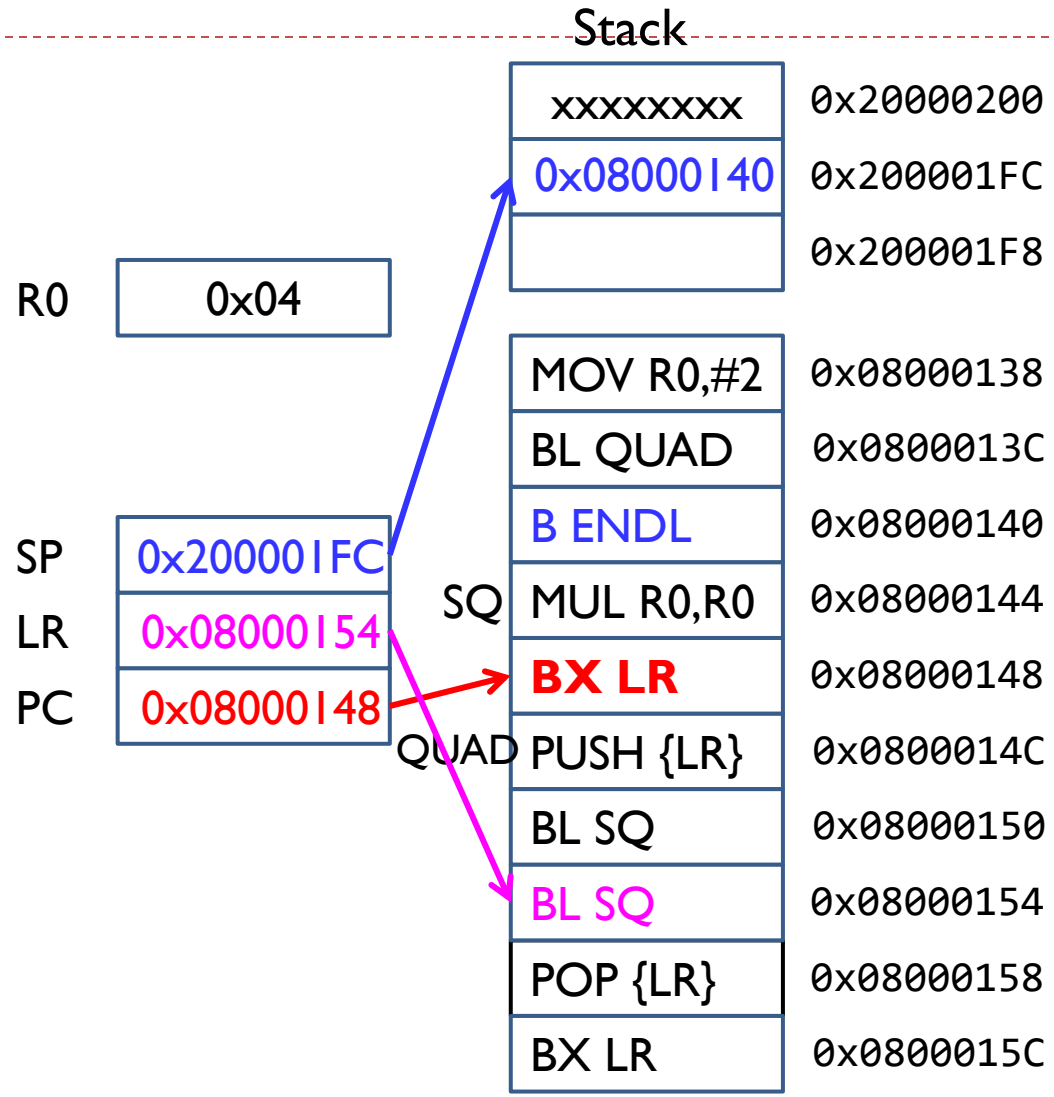


Example: R0 = R0^4

```
        MOV R0,#2
        BL QUAD
        B ENDL

SQ      MUL R0,R0
        BX LR

QUAD    PUSH {LR}
        BL SQ
        BL SQ
        POP {LR}
        BX LR
ENDL    ...
```

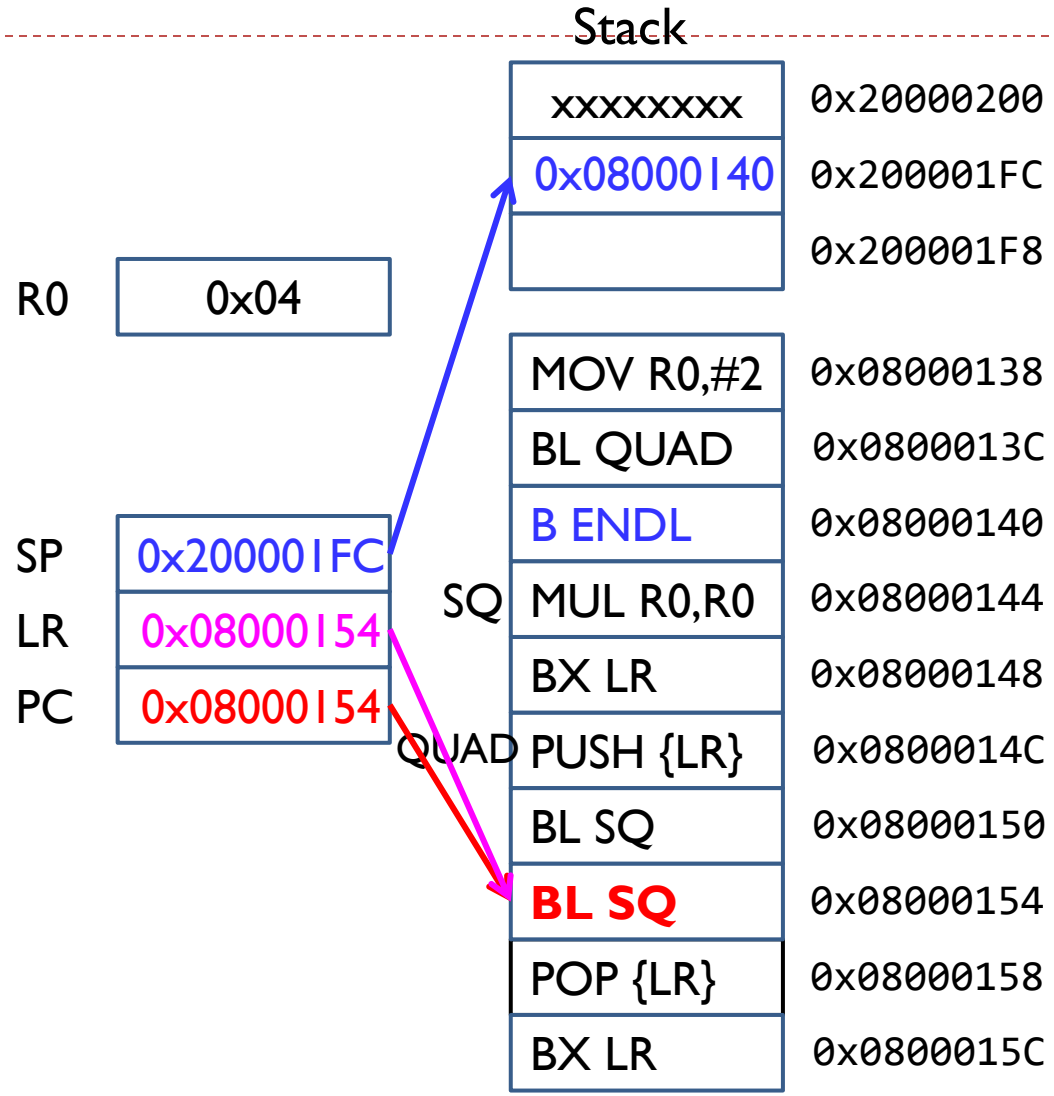


Example: R0 = R0^4

```
        MOV R0,#2
        BL QUAD
        B ENDL

SQ      MUL R0,R0
        BX LR

QUAD    PUSH {LR}
        BL SQ
        BL SQ
        POP {LR}
        BX LR
ENDL    ...
```

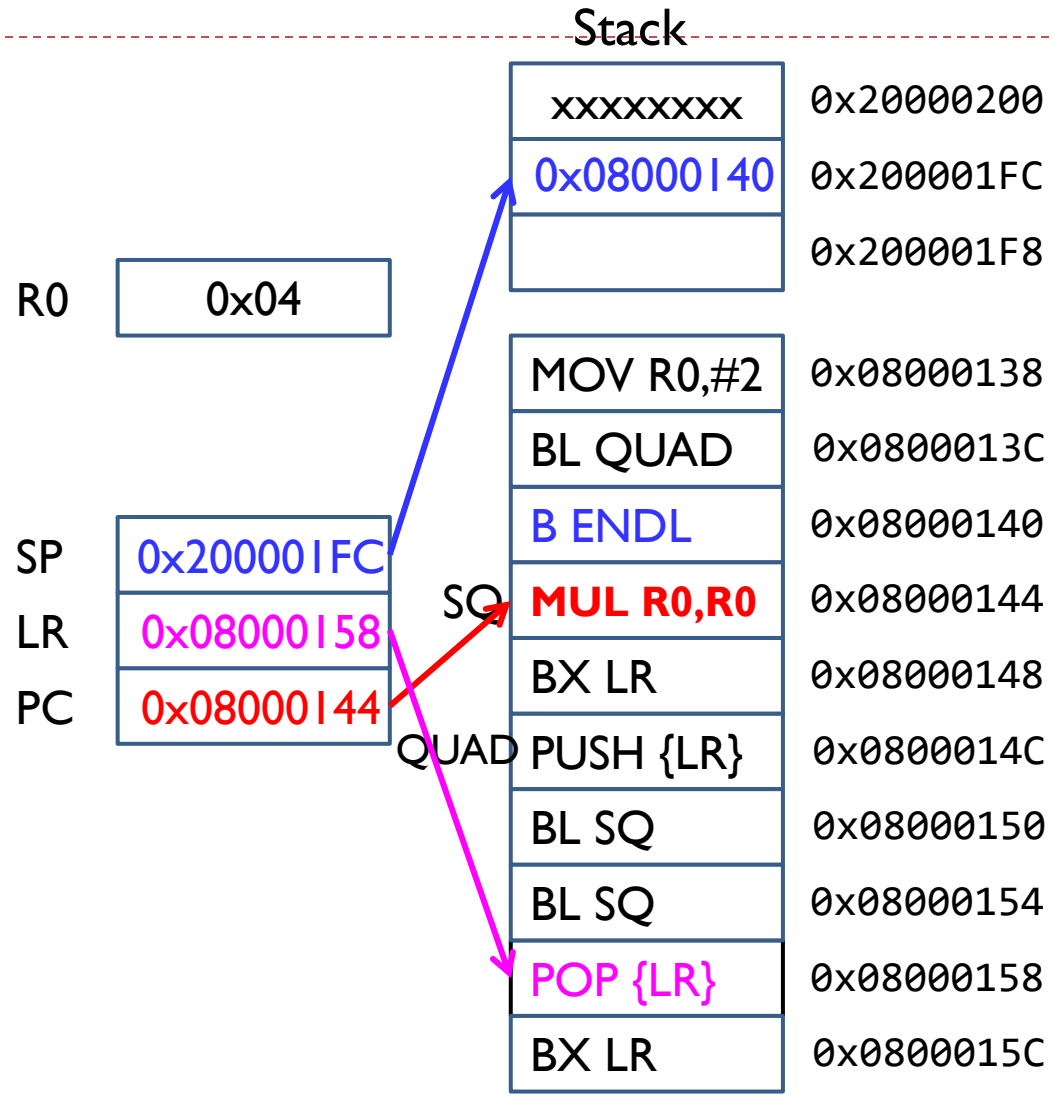


Example: R0 = R0^4

```
        MOV R0,#2
        BL QUAD
        B ENDL

SQ      MUL R0,R0
        BX LR

QUAD    PUSH {LR}
        BL SQ
        BL SQ
        POP {LR}
        BX LR
ENDL    ...
```

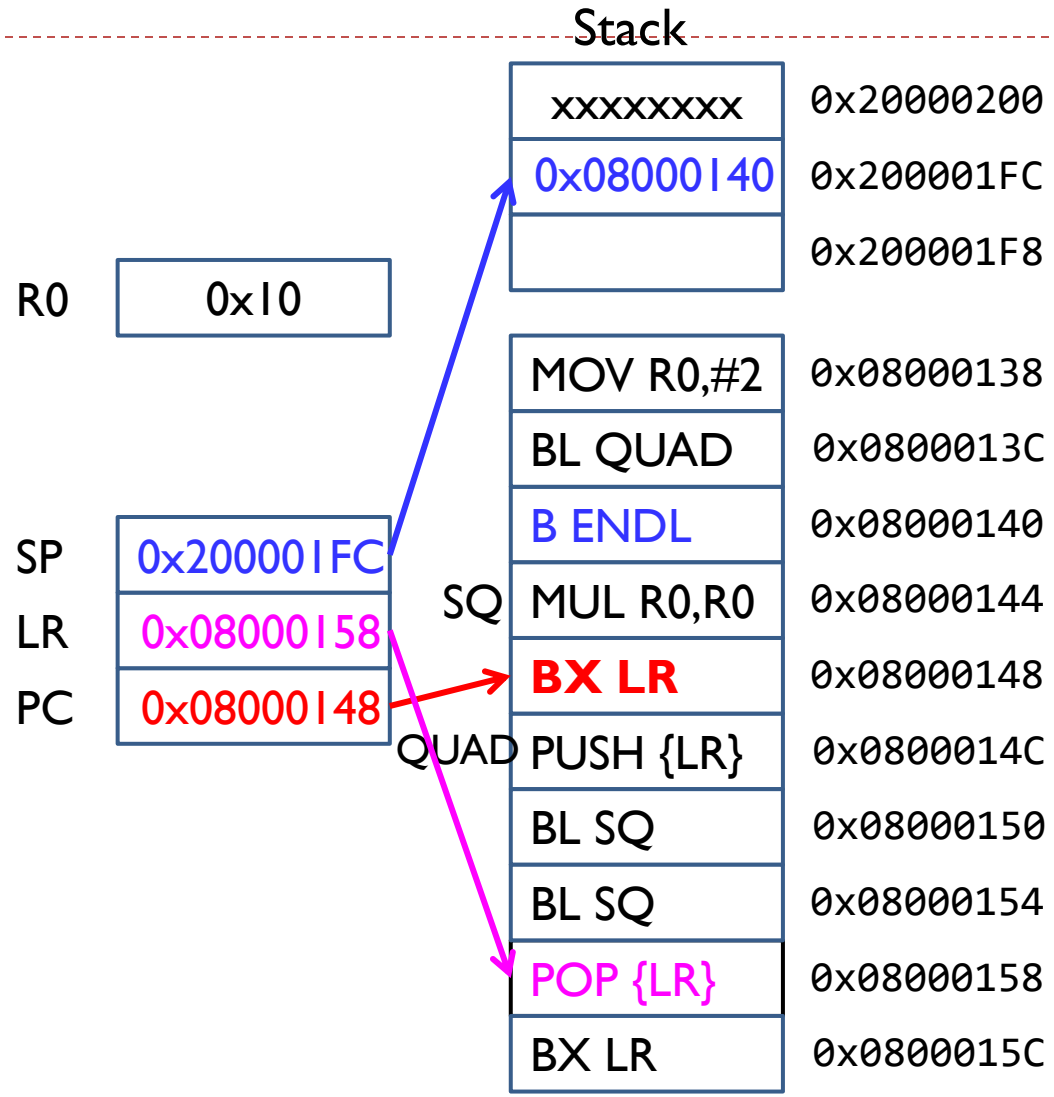


Example: R0 = R0⁴

```
        MOV R0,#2
        BL QUAD
        B ENDL

SQ      MUL R0,R0
        BX LR

QUAD    PUSH {LR}
        BL SQ
        BL SQ
        POP {LR}
        BX LR
ENDL    ...
```

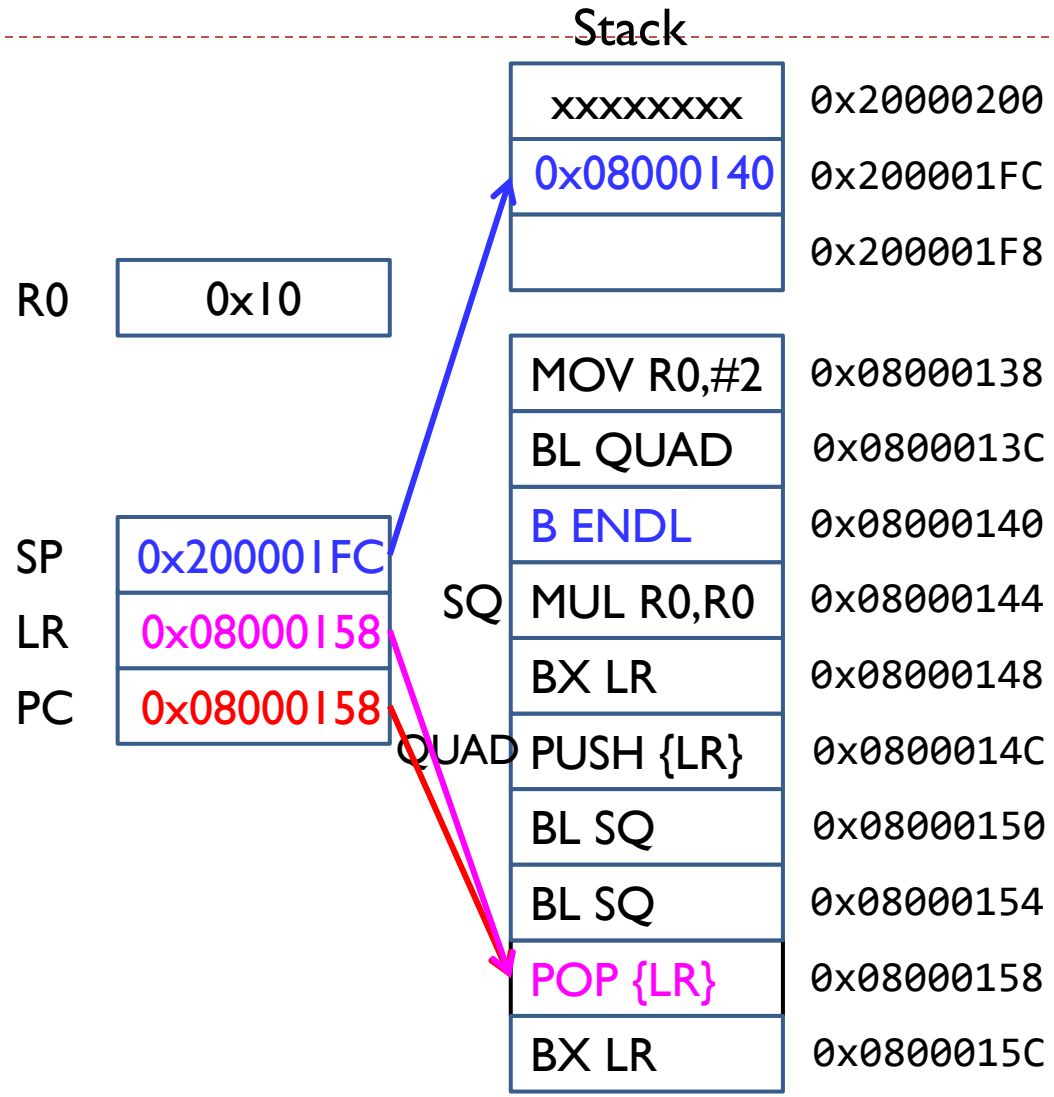


Example: R0 = R0⁴

```
        MOV R0,#2
        BL QUAD
        B ENDL

SQ      MUL R0,R0
        BX LR

QUAD    PUSH {LR}
        BL SQ
        BL SQ
        POP {LR}
        BX LR
ENDL    ...
```

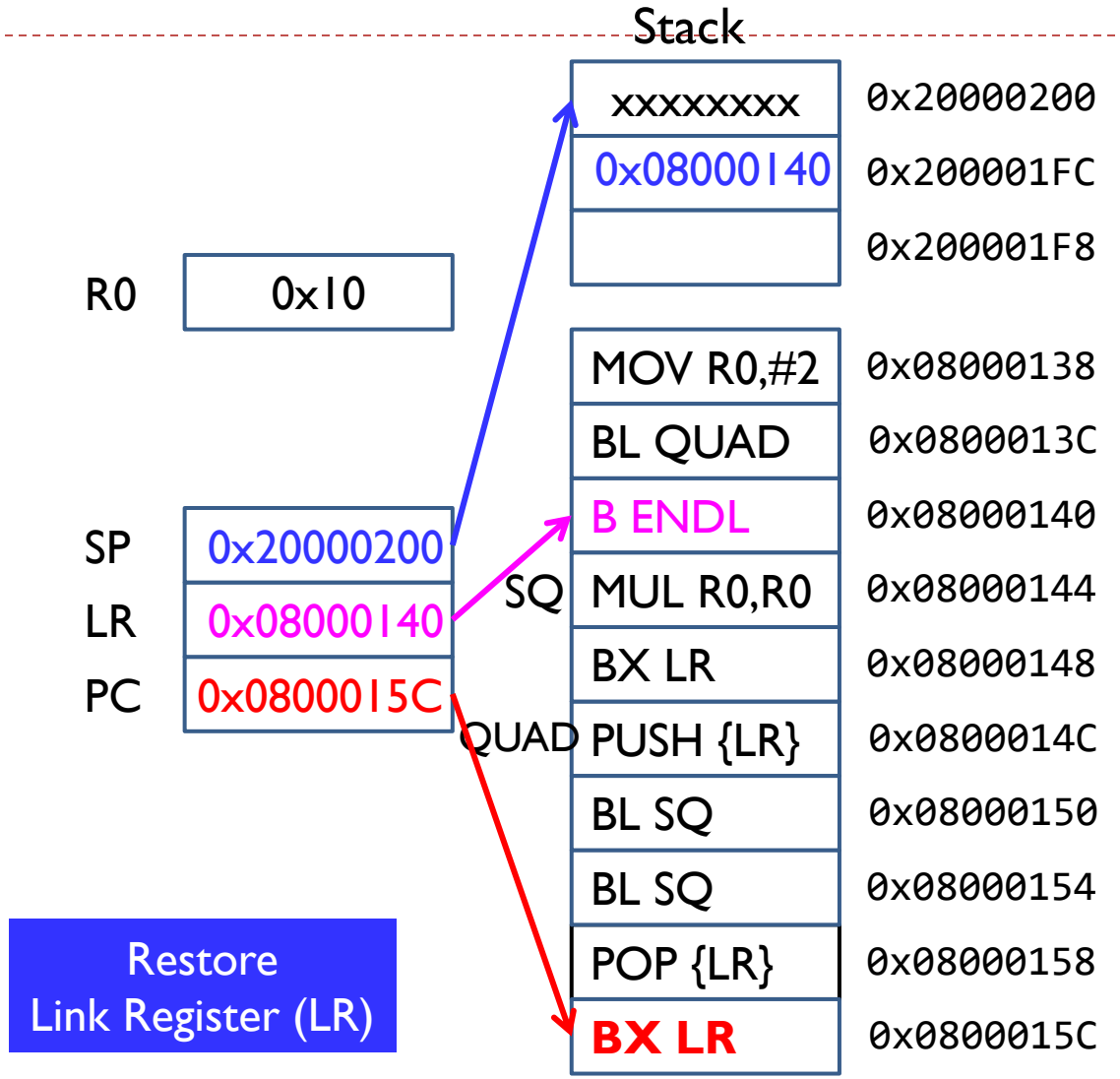


Example: R0 = R0^4

```
        MOV R0,#2
        BL QUAD
        B ENDL

SQ      MUL R0,R0
        BX LR

QUAD    PUSH {LR}
        BL SQ
        BL SQ
        POP {LR}
        BX LR
ENDL    ...
```

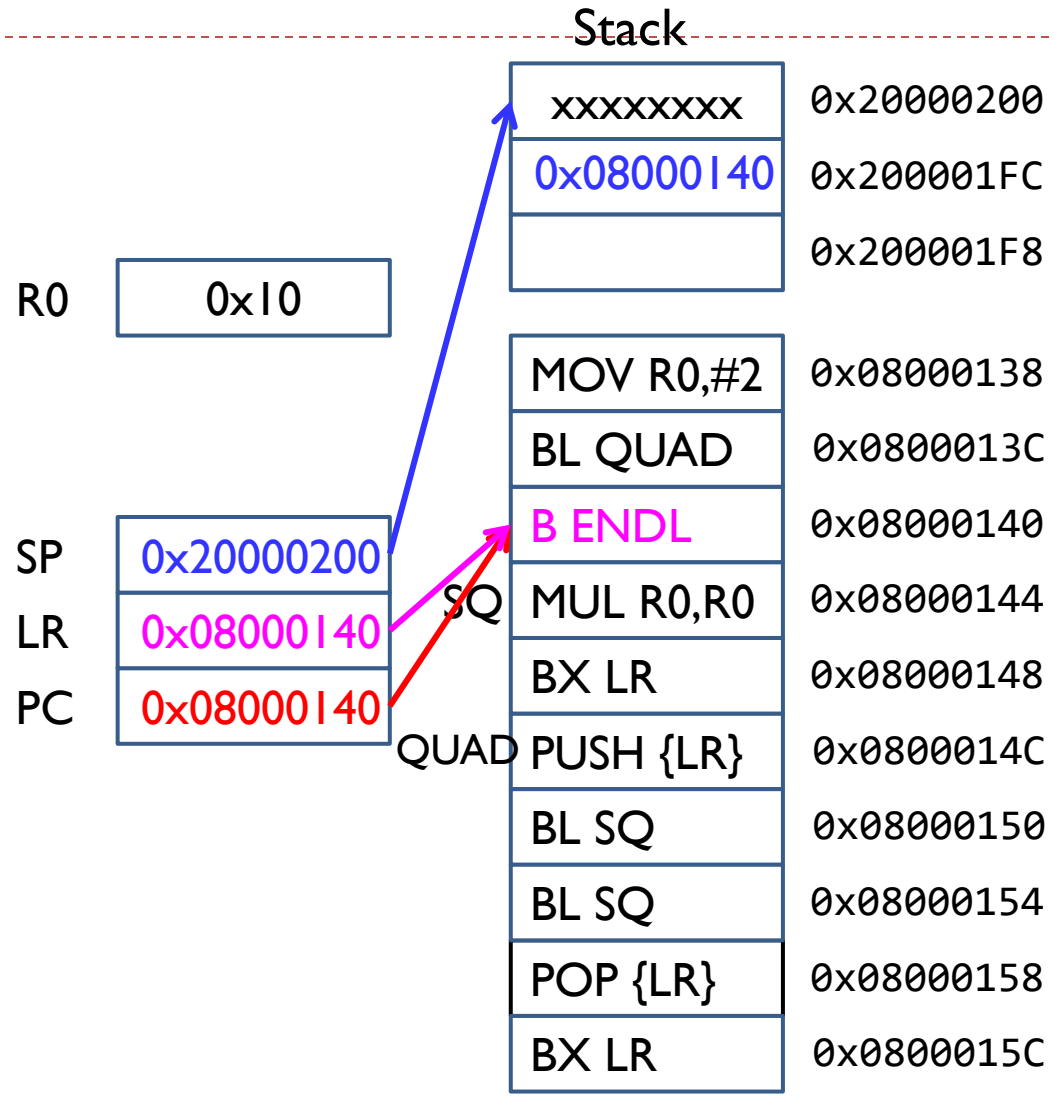


Example: R0 = R0^4

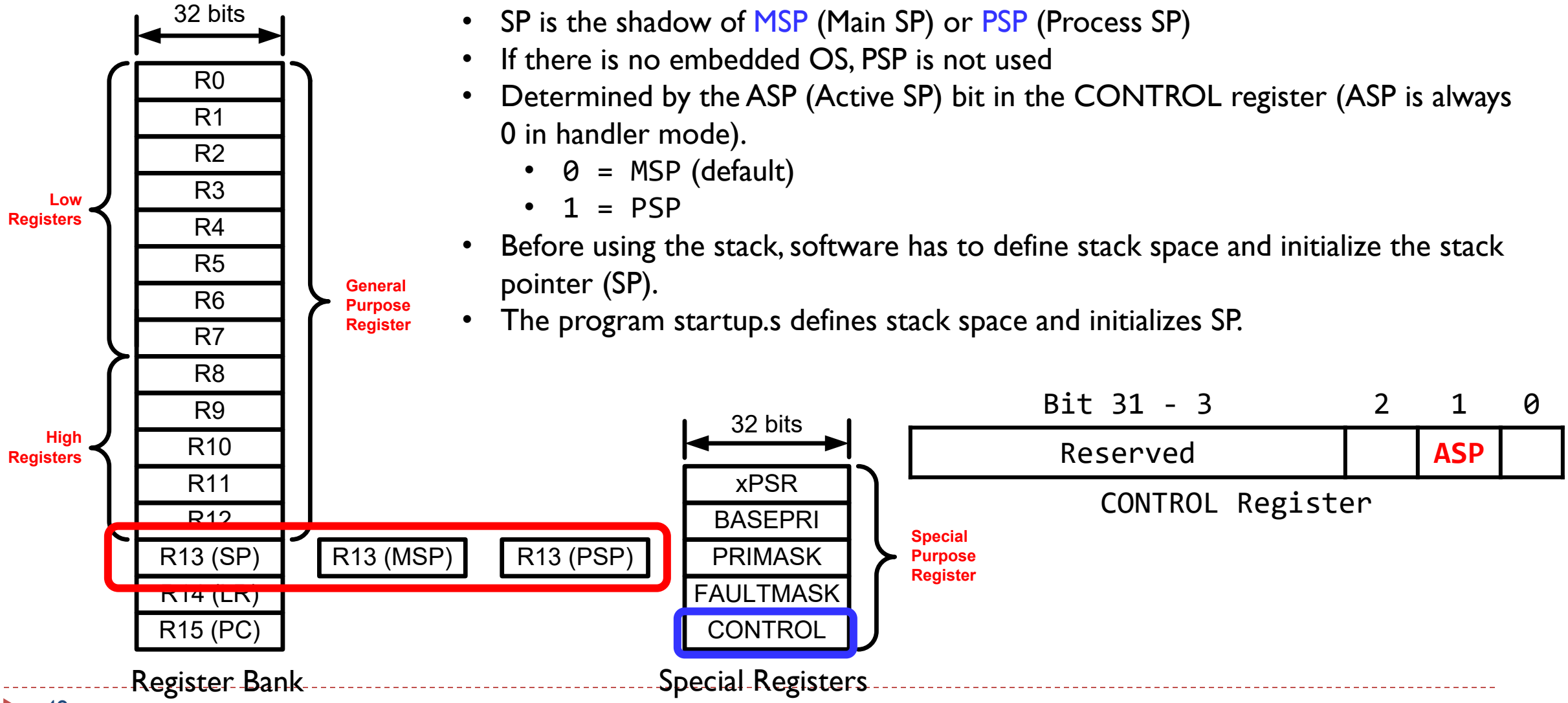
```
        MOV R0,#2
        BL QUAD
        B ENDL

SQ      MUL R0,R0
        BX LR

QUAD    PUSH {LR}
        BL SQ
        BL SQ
        POP {LR}
        BX LR
ENDL    ...
```



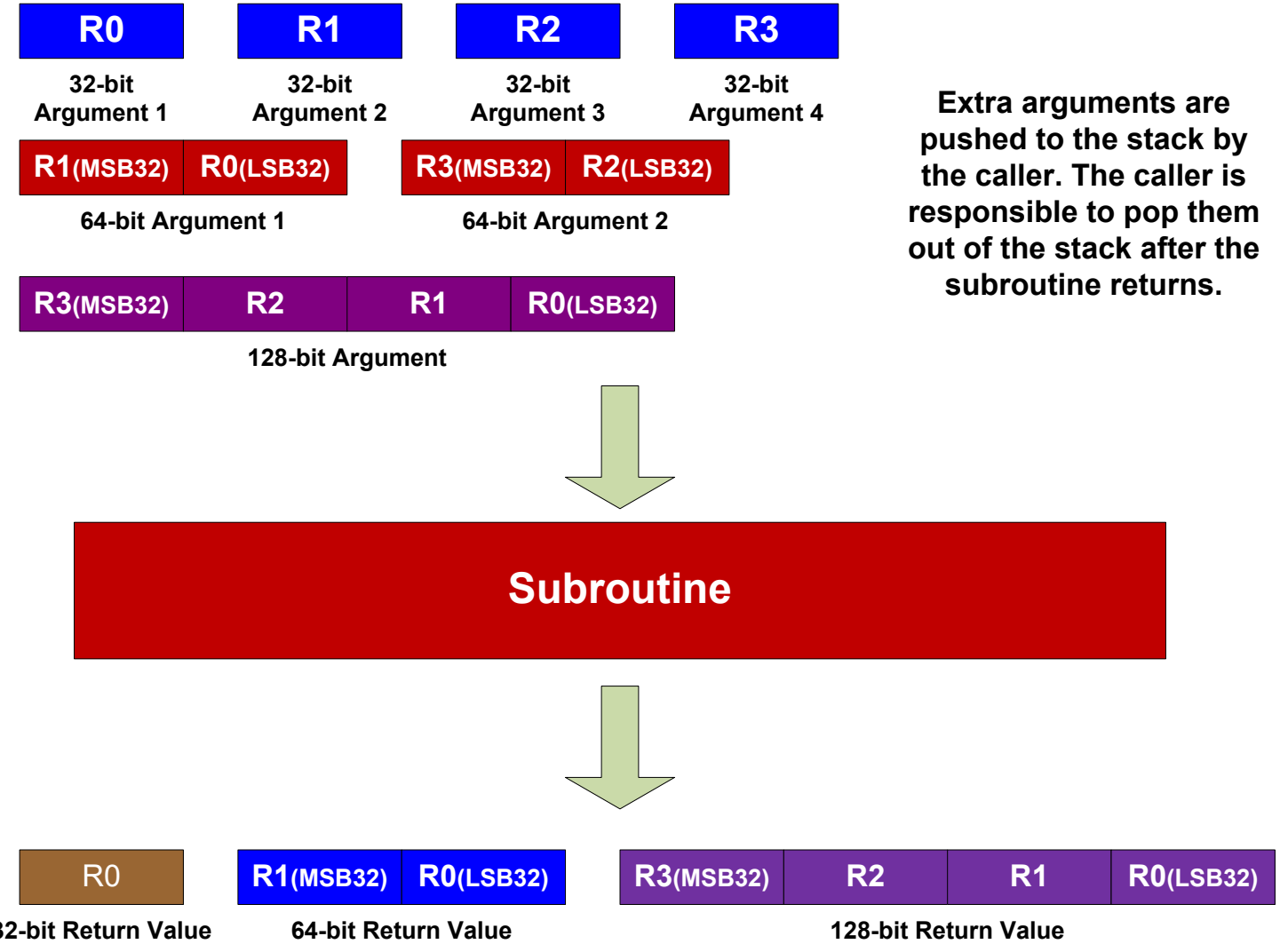
Stack Pointer (SP)



- SP is the shadow of MSP (Main SP) or PSP (Process SP)
- If there is no embedded OS, PSP is not used
- Determined by the ASP (Active SP) bit in the CONTROL register (ASP is always 0 in handler mode).
  - `0` = MSP (default)
  - `1` = PSP
- Before using the stack, software has to define stack space and initialize the stack pointer (SP).
- The program startup.s defines stack space and initializes SP.

Review

Passing Arguments into a Subroutine

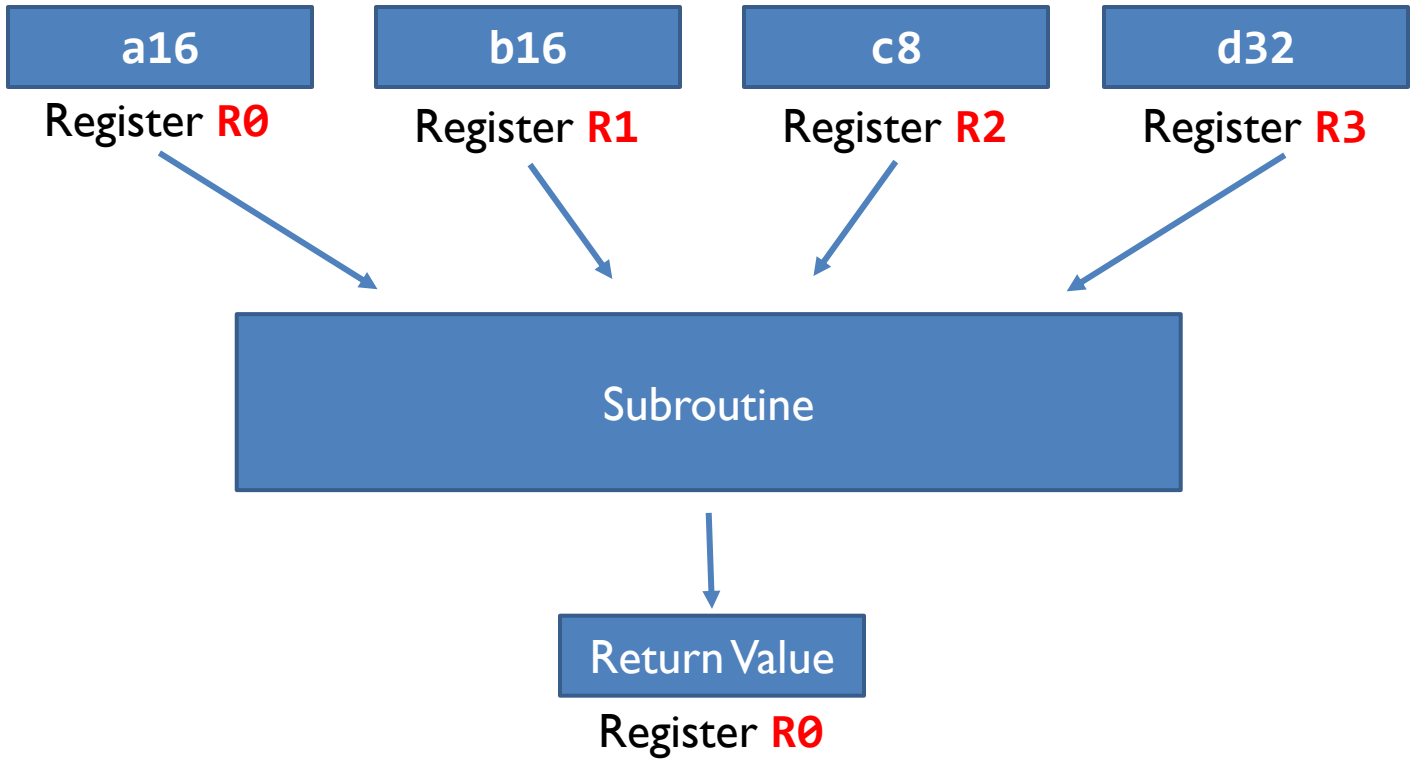


Extra arguments are pushed to the stack by the caller. The caller is responsible to pop them out of the stack after the subroutine returns.

Review

Passing Arguments into a Subroutine

```
int32_t sum(int16_t a16, int16_t b16, int8_t c8, int32_t d32);
```



Review

Passing 4 Arguments

```
int32_t sum(int16_t a16, int16_t b16, int8_t c8, int32_t d32);
```

```
s = sum(1, 2, 3, 4);
```

Caller

```
  MOVS r0, #1 ; a16
  MOVS r1, #2 ; b16
  MOVS r2, #3 ; c8
  MOVS r3, #4 ; d32
  BL   sum
```

Callee

```
sum PROC
  ADD r0, r0, r1 ; a16 + b16
  ADD r0, r0, r2 ; add c8
  ADD r0, r0, r3 ; add d32
  BX  LR         ; return
  ENDP
```

Passing Extra Arguments via Stack

```
int32_t sum(int32_t a, int32_t b, int32_t c,
int32_t d, int32_t h, int32_t i, int32_t j,
int32_t k);
s = sum(1, 2, 3, 4, 5, 6, 7, 8);
```

Caller

```
MOVS r0, #5
MOVS r1, #6
MOVS r2, #7
MOVS r3, #8
PUSH {r0, r1, r2, r3}
MOVS r0, #1
MOVS r1, #2
MOVS r2, #3
MOVS r3, #4
BL   sum
...
POP {r0, r1, r2, r3}
```

Version 1

```
sum PROC
  EXPORT sum
  ADD r0, r0, r1    ; add a + b
  ADD r0, r0, r2    ; add c
  ADD r0, r0, r3    ; add d
  LDRD r1,r2, [sp] ; r1=mem[sp],r2=mem[sp+4]
  ADD r0, r0, r1    ; add h
  ADD r0, r0, r2    ; add i
  LDRD r1,r2, [sp, #8] ; r1=mem[sp+8],r2=mem[sp+12]
  ADD r0, r0, r1    ; add j
  ADD r0, r0, r2    ; add k
  BX  LR
  ENDP
```

Version 2

```
sum PROC
  EXPORT sum
  PUSH {r5, r6, lr}
  ADD r0, r0, r1    ; add a + b
  ADD r0, r0, r2    ; add c
  ADD r0, r0, r3    ; add d
  LDRD r5,r6, [sp, #12] ;r5=mem[sp+12],r6=mem[sp+16]
  ADD r0, r0, r5    ; add h
  ADD r0, r0, r6    ; add i
  LDRD r5,r6, [sp, #20] ;r5=mem[sp+20],r6=mem[sp+24]
  ADD r0, r0, r5    ; add j
  ADD r0, r0, r6    ; add k
  POP {r5, r6, pc}
  ENDP
```

Explanations

- **Version 1:**
- Callee reads the extra args directly from the caller's push with LDRD: Load Register Doubleword:
  - LDRD r1,r2, [sp] → loads 5th & 6th args (h, i)
  - LDRD r1,r2, [sp,#8] → loads 7th & 8th args (j, k)
- It returns with BX LR.
- **Version 2:**
- Callee **pushes** r5, r6, and lr on entry: PUSH {r5, r6, lr}.
- Because it pushed, the extra-argument addresses are shifted, so it uses offsets like [sp,#12] and [sp,#20] to read the caller's arguments into r5,r6.
  - LDRD r5,r6, [sp, #12]
  - LDRD r5,r6, [sp, #20]
- It returns via POP {r5, r6, pc} (popping lr into pc returns directly).
- Ladder analogy:
  - Think of the stack like a ladder: The ladder = one shared stack in memory.
  - The rung each function stands on = its current SP value.
  - When a function "pushes," it steps down a few rungs (SP decreases).
  - The caller's pushed data remains up above — just higher up the same ladder.

Address	Contents	Access
0x0FFC	arg8 (k=8)	[sp,#12]
0x0FF8	arg7 (j=7)	[sp,#8]
0x0FF4	arg6 (i=6)	[sp,#4]
0x0FF0	arg5 (h=5)	[sp] ← SP

Version 1 stack after Caller PUSH

Address	Contents	Access
0x0FFC	arg8 (k=8)	[sp,#24]
0x0FF8	arg7 (j=7)	[sp,#20]
0x0FF4	arg6 (i=6)	[sp,#16]
0x0FF0	arg5 (h=5)	[sp,#12]
0x0FEC	saved lr	[sp,#8]
0x0FE8	saved r6	[sp,#4]
0x0FE4	saved r5	[sp] ← SP

Version 2 stack after Caller PUSH and Callee PUSH

Version 2 is Better Programming Practice

- Caller pushes extra arguments (5th to 8th) onto the stack before calling the function.
- Version 1: Callee accesses those extra arguments directly from the stack using LDRD instructions at specific offsets from SP. It demonstrates the basic mechanism for handling extra arguments on the stack.
- Version 2: Callee begins by pushing registers r5, r6, and the link register (LR) (the callee-saved registers) onto the stack to preserve them. After addition operations, it pops r5, r6, and the program counter (PC) to return, restoring the preserved registers and the return address. This ensures that the subroutine does not unintentionally overwrite or lose the caller's data and return address, maintaining program correctness during and after the function call.

Summary

- ARM Cortex-M uses full descending stack
- How to pass arguments into a subroutine?
  - Each 8-, 16- or 32-bit parameter is passed via `r0`, `r1`, `r2`, `r3`
  - Extra parameters are passed via the stack
- What registers should be preserved?
  - Caller-saved registers *vs* callee-saved registers
- How to preserve the running environment for the caller?
  - Via stack

References

- Lecture 31. Preserving registers in a Subroutine
  - https://www.youtube.com/watch?v=DGKjFKjxAYs&list=PLRJhV4hUhIymmp5CCeIFPyxbknsdcXCc8&index=31